Estrategias para la implementación de algoritmos de satisfactibilidad en lógica de reescritura

Universidad Complutense de Madrid
Facultad de Matemáticas

Trabajo de fin de grado del Doble Grado en Ingeniería Informática y Matemáticas

Alejandro Hernández Cerezo

Dirigido por:
Narciso Martí Oliet
José Alberto Verdejo López

Curso 2019-2020

Madrid, 20 de junio de 2020

Resumen

El problema de satisfactibilidad booleana (SAT) consiste en determinar si existe o no una asignación válida de variables que satisfaga un conjunto de fórmulas lógicas. Este problema es un problema clásico de las ciencias de la computación, siendo el primer problema catalogado como NP-completo. Además, tiene multitud de aplicaciones tanto en otros campos de estudio como a nivel industrial.

La mayoría de resolutores SAT actuales se basan en el cálculo de David-Putnam-Logemann-Loveland (DPLL), un sistema de reglas completo que certifica poder encontrar una asignación válida en el caso de que exista. Existen otros cálculos más avanzados, como el cálculo de Tinelli, que modelan algunas de las técnicas avanzadas que se utilizan en los resolutores actuales: *backjump*, aprendizaje de cláusulas o reinicios selectivos. Otras técnicas consisten en aplicar de manera razonada las reglas de inferencia conforme a ciertas heurísticas, por lo que se necesita una separación clara entre el sistema de reglas y las heurísticas.

La lógica de reescritura es un marco ideal para estudiar estas técnicas. En ella, se hace una separación explícita en 3 niveles: ecuaciones, en donde podemos definir las operaciones y predicados auxiliares; reglas, en el que se incluyen las reglas de inferencia; y estrategias, donde controlamos la aplicación de reglas conforme a estas heurísticas.

El objetivo de este trabajo es adaptar las reglas del cálculo de Tinelli al sistema Maude, refinando y extendiendo algunas de sus declaraciones. Se incluirán las técnicas de resolutores SAT actuales mencionadas anteriormente, así como una comparación de distintas heurísticas definidas en algunos de los resolutores más famosos. Para ello, estudiaremos en detalle y utilizaremos el lenguaje de estrategias de Maude, una funcionalidad menos conocida de este sistema que permite aplicar las reglas de una forma controlada y eficiente. Además, se realizarán distintos experimentos comparando la eficiencia de distintas estrategias.

Palabras clave

Backjump, Conflict-driven clause learning, Algoritmo DPLL, Estrategias, Lógica de reescritura, Maude, Unique Implication Point, SAT

Abstract

The Boolean satisfiability problem (SAT) consists of determining whether there exists an assignment that satisfies a given Boolean formula. SAT is a classic computer science problem, as it was the first problem proven to be NP-complete. SAT has also many applications in different fields.

Most modern SAT solvers are based on the David-Putnam-Logemann-Loveland algorithm (DPLL), a complete backtracking-based search algorithm that manages to find a valid assignment in case it exists. Tinelli's inference system extends this version to include other modern features, such as backjump, clause learning and restarts. Modern SAT solvers also include heuristics, so we are interested in studying both rules and heuristics to understand how they work. There is usually no clear separation between these two levels, making it difficult to understand both aspects.

Rewriting logic is an interesting framework to study these techniques. Declarations can be included in three different levels: equations, rules and strategies. The equation level includes declarations of logical symbols and auxiliary predicates which are needed to declare the rules in the inference system. These rules are declared in the rule level. The strategy level contains statements to control how rules are applied, so heuristics are represented in this level.

The goal of this work is to study different modern SAT solvers' techniques by adapting them to rewriting logic. For this purpose, we will implement Tinelli's framework into the Maude system, extending its declarations to include other techniques. Therefore, we will study in detail how these techniques and heuristics work, and how to implement them in Maude. This project also aims to show the capabilities of the strategy language, which is a functionality recently released in Maude for controlling the application of rules. Furthermore, we will perform experiments to compare the performance of these techniques.

Keywords

Backjump, Conflict-driven clause learning, DPLL algorithm, Strategies, Rewriting Logic, Maude, Unique Implication Point, SAT

Índice general

Índice de figuras

Capítulo 1

Introducción

El propósito de este trabajo es estudiar el problema de satisfactibilidad booleana (SAT), de tal forma que podamos proponer una abstracción adecuada de distintos sistemas de inferencia que lo resuelvan, utilizando lógica de reescritura. En este capítulo vamos a introducir estos conceptos claves y explicar cómo encajan entre sí, incluyendo también un resumen de todas las aportaciones realizadas.

Introducimos el problema del SAT y comentamos algunas formas propuestas de resolverlo en la sección 1.1. Continuamos explicando el marco de la lógica de reescritura en la sección 1.2 e introduciendo el lenguaje Maude en la sección 1.3. La sección 1.4 aborda más directamente los objetivos que se persiguen con este trabajo, incluyendo las distintas técnicas y heurísticas que hemos analizado y en qué secciones de la memoria se detallan. Por último, en la sección 1.5 comentamos la planificación que se ha seguido para realizar este trabajo.

1.1. El problema de satisfactibilidad booleana

La lógica ha sido uno de los grandes pilares del conocimiento humano desde tiempos inmemoriables. Muchas civilizaciones han contribuido al desarrollo del pensamiento lógico y del silogismo, siendo especialmente notables las aportaciones de las civilizaciones griega, india y china. Entre ellas, destacan los trabajos de Aristóteles, a quién se considera uno de los grandes precursores de la lógica en Occidente. En sus trabajos, se hizo un estudio pormenorizado de los principios del razonamiento y del silogismo. Estos trabajos tuvieron un gran impacto en la época medieval, siendo ampliamente estudiados durante la Edad Media por parte de los filósofos de la época.

Por otro lado, los estoicos fueron los primeros en introducir la lógica proposicional, que difería bastante de la lógica de Aristóteles. Sin embargo, no fue hasta principios del siglo XIX cuando surgen los primeros trabajos que estudian la lógica proposicional de manera formal. Algunos de los mayores avances vinieron dados por matemáticos como George Boole y Augustus de Morgan, y ampliados por muchos otros científicos y filósofos.

El problema de satisfactibilidad booleana (SAT) tal y como lo conocemos hoy en día cobró una mayor relevancia a partir de la introducción del estudio de las ciencias de la computación. Este problema consiste en averiguar si dada una serie de fórmulas en lógica proposicional, existe alguna asignación de sus variables que hace que se cumplan todas ellas.

El problema del SAT demostró ser un problema crucial en el estudio de la complejidad computacional. Para entender el porqué de su importancia, vamos a hacer una breve introducción a la teoría de la complejidad computacional.

Un problema se dice que pertenece a P si se puede resolver con un algoritmo en tiempo polinómico.

Un problema se dice que pertenece a NP si se puede resolver con una máquina de Turing no determinista en tiempo polinómico. Esto es equivalente a afirmar que dado un problema y una posible solución al mismo, se puede verificar la solución con complejidad polinómica.

Es claro que $P \subseteq NP$, pero no se sabe si $NP \subseteq P$, o equivalentemente, si $P = NP$. Este es uno de los grandes problemas matemáticos abiertos hoy en día, que podría tener un impacto bestial en el mundo en el que vivimos si se llega a probar efectivamente que $P = NP$.

Existe un tipo especial de problemas NP, los llamados problemas NP-completos. Estos problemas son un subconjunto de los problemas NP, que verifican que cualquier problema NP se puede reducir a uno NP-completo en tiempo polinómico. Por tanto, bastaría con probar que un problema NP-completo está en P para probar la conjetura $P = NP$.

Existe una gran lista de problemas NP-completos, la mayoría de los cuáles se han incluido en esta categoría a partir de la reducción a otro problema NP-completo en tiempo polinómico. El primer problema que se descubrió que era NP-completo fue el problema del SAT, cuyo enunciado corresponde al Teorema de Cook. Este teorema se basa en la transformación de cualquier máquina de Turing no determinista en una serie de fórmulas booleanas, de manera que se puede simular su comportamiento.

El problema del SAT no solo tiene una fuerte presencia en el estudio de la complejidad computacional. Tiene multitud de aplicaciones en campos muy estudiados actualmente como la Inteligencia Artificial, Investigación Operativa, Verificación o Diseño Automático. También algunos campos de la industria están empezando a aplicar resolutores SAT en sus procesos, pues los resolutores actuales son capaces de resolver problemas con miles de variables de decisión y millones de cláusulas.

Una de las extensiones más famosas del SAT es el problema de satisfactibilidad módulo teorías (SMT), que consiste en decidir si un conjunto de fórmulas lógicas de primer orden son satisfactibles de acuerdo a una teoría subyacente. Esta aproximación permite expresiones más ricas, pudiendo modelar y resolver problemas más complejos que los englobados por SAT.

Tanto SAT como SMT están fuertemente relacionados con un paradigma de programación: la programación con restricciones. Este paradigma difiere del paradigma de programación imperativa en que no se especifica cómo resolver el problema paso a paso, sino que se fijan una serie de restricciones para un conjunto de variables en un dominio definido, y se elige un método general que lo resuelva.

SAT puede estar presentado en distintas variantes, dependiendo del tamaño de las cláusulas o la forma en que estas aparecen. La presentación del problema tiene un gran impacto en la complejidad del mismo. Así, por ejemplo, si se presentan las cláusulas en su forma normal disyuntiva (DNF), determinar si una instancia es satisfactible o no se puede realizar en tiempo lineal. O en el caso de que todas las cláusulas tengan tamaño 2 (2-SAT), el problema del SAT ya no es NP-completo, sino que existen algoritmos que pueden determinar en tiempo lineal si existe solución o no.

Nosotros vamos a trabajar con la versión más común del problema del SAT: el conjunto de cláusulas vendrán dadas en su forma normal conjuntiva (CNF). Además, vamos a trabajar con instancias del problema del 3-SAT, siguiendo algunos de los experimentos estandarizados que se emplean en competiciones como [10]. Estos experimentos se detallan en el Capítulo 7.

Uno de los sistemas más utilizados para estudiar el problema del CNF-SAT es el algoritmo de David-Putnam-Logemann-Loveland (DPLL). Consiste en un algoritmo de búsqueda completo basado en vuelta atrás, definiendo para ello una serie de reglas de inferencia. Otros trabajos recientes han extendido estas reglas para estudiar el problema de SMT. En la sección 2.2 las estudiaremos en más detalle.

En nuestro caso, vamos a centrarnos en estudiar las ecuaciones de Tinelli, que vienen recogidas en distintas versiones en los trabajos [9] y [13]. Estas ecuaciones son más interesantes desde el punto de vista de la eficiencia, pues tienen en cuenta algunas técnicas de SAT modernas como son *Backjump*, aprendizaje de cláusulas o reinicios selectivos. En la sección 3.1 introduciremos estas reglas con alguna ligera modificación que las hace más adecuadas para su posterior implementación. Explicaremos además en qué consisten estas técnicas y cómo adaptarlas a nivel de implementación con reglas.

Estas técnicas se engloban dentro del algoritmo conocido como aprendizaje de cláusulas dirigido por conflictos, o *conflict-driven clause learning* (CDCL). La idea básica de este enfoque consiste en aplicar las reglas de DPLL siguiendo un orden preestablecido para intentar generar conflictos lo antes posible. Cada vez que se elige una asignación de una variable, se comprueba si existe alguna otra asignación que se pueda deducir de esta, y así sucesivamente hasta no poder elegir una nueva variable o llegar a alguna contradicción. Una vez se llega a una contradicción con las asignaciones actuales, se analizan las cláusulas previas que han conducido a ella para así generar una cláusula de *backjump*. Esta cláusula sirve tanto para deshacer varias decisiones de una vez, como para incluirla en el conjunto de cláusulas a tratar.

Muchos de los resolutores SAT modernos siguen este algoritmo: Chaff [8], Berkmin [5], PicoSat [1], MiniSat [4]... Difieren unos de otros esencialmente en las siguientes cuestiones:

1. ¿Cómo se decide cuál es la siguiente variable a asignar? ¿Le asignamos valor 0 o 1?
2. ¿Cómo se genera la cláusula de backjump óptima?
3. ¿Cómo comprobamos si se deduce alguna asignación nueva a partir de una variable previa que hemos asignado?
4. ¿Qué política se sigue para eliminar cláusulas previamente aprendidas para que el sistema no se desborde?
5. ¿Cómo se sale de una exploración infructuosa, y se exploran nuevas opciones?

En nuestro caso, vamos a intentar dar respuesta a todas esas cuestiones por cada uno de los casos que analicemos en los siguientes capítulos.

1.2. Lógica de reescritura

Gran parte de la explicación de este apartado consiste en un resumen de la explicación de lógica de reescritura en el documento [7]. Lo hemos simplificado para quedarnos con aquellas partes que tienen impacto en este proyecto.

La lógica de reescritura es un marco que permite expresar de forma natural y general tanto las especificaciones semánticas de modelos de concurrencia, sistemas distribuidos o lenguajes de programación, como distintas lógicas y sistemas de inferencia. Este comportamiento se consigue a través de la introducción de *teorías de reescritura*, que consisten en una tupla $\mathcal{R} = (\Sigma, E, R)$. En esta expresión, el par (Σ, E) representa una teoría ecuacional y R las reglas de reescritura.

La lógica de reescritura se puede entender desde un punto de vista lógico o computacional. En nuestro caso, el estudio del problema del SAT se entiende mejor desde un punto de vista lógico, así que nos vamos a centrar en explicar la especificación anterior de acuerdo a esta visión. En esta visión, se entiende que los estados que estamos considerando corresponden a fórmulas.

La colección Σ representa el conjunto de símbolos lógicos y no lógicos que se utiliza para construir una fórmula, así como los operadores para construirlas. A su vez, el conjunto E representa las identidades algebraicas existentes entre los elementos construidos sobre Σ. De esta forma, cada elemento de nuestra colección pertenece a una clase de equivalencia inducida por E. Podemos decir que dos elementos t y t' sobre Σ cumplen la igualdad $t = t'$ si al reducirlos utilizando las ecuaciones en E llegamos al mismo término final.

El conjunto R describe las reglas de inferencia de esa lógica. Esto no implica que necesariamente se reescriban únicamente fórmulas, sino que también se podrían aplicar estas reglas a otras estructuras basadas en fórmulas.

Las reglas de reescritura son generalmente de la forma $t \mapsto t'$, donde pasamos de una expresión algebraica que depende de t a otra que depende de t'. En la expresión de la regla, el término izquierdo corresponde al término que activa la regla, mientras que el derecho corresponde a la sustitución final.

Las expresiones t y t' no representan necesariamente términos concretos de nuestra colección, sino que pueden representar patrones paramétricos que describen familias de elementos sobre Σ. Estas pueden ser instanciadas a términos concretos a través de una sustitución θ.

La manera de proceder al aplicar las reglas de reescritura es bastante sencilla: primero se encuentra un subtérmino que encaje con t a partir de una sustitución θ, y luego se sustituye con $\theta(t')$.

Trabajar con teorías de reescritura nos permite representar estructuras tan complejas como queramos de una manera natural. Esto se debe a que la propia especificación de los sistemas en lógica de reescritura es muy cercana a la especificación algebraica formal, por lo que la diferencia entre una representación en lógica de reescritura y su formalismo es casi inexistente.

También es importante tener en cuenta que la diferenciación entre ecuaciones y reglas nos permite distinguir las partes estáticas y dinámicas de nuestra representación. Las partes estáticas se corresponden con la teoría ecuacional (Σ, E), mientras que la parte dinámica se corresponde con las reglas de reescritura R.

Esta distinción va a ser clave para este trabajo, pues surge de forma natural en el problema del SAT. Así, las reglas de reescritura vendrían conformadas por las reglas del sistema de inferencia, que esencialmente se basan en realizar y deshacer dinámicamente decisiones según las condiciones del momento. Por otro lado, la teoría ecuacional correspondería a las definiciones, construcción de fórmulas y predicados que nos ayudan a hacer comprobaciones para posteriormente aplicar las reglas.

Este apartado probablemente sea el más tedioso del trabajo, pues al tratarse de un concepto bastante abstracto, es complicado entenderlo sin llevarlo a ejemplos concretos. No obstante, el resto de capítulos se pueden entender perfectamente sin el uso del mismo. Igualmente, se recomienda tener cierto entendimiento de estas nociones para entender el lenguaje Maude.

1.3. El lenguaje Maude

En esta sección, vamos a hacer una breve introducción al lenguaje Maude y comentar algunas de sus características más relevantes. Hemos incluido en el apéndice A una explicación más detallada de

todas los aspectos de Maude que necesitamos considerar a la hora de entender la implementación de esta memoria.

Maude es un lenguaje de alto nivel y una herramienta basada en lógica de escritura, que permite ejecutar, analizar y verificar la especificación de sistemas. Maude sirve tanto de marco semántico para definir semánticas de lenguajes y modelos de concurrencia, como marco lógico donde expresar lógicas e implementarlas. Maude admite reflexión, es decir, permite modificar su estructura de alto nivel y trabajar con opciones de metaprogramación.

Maude destaca por su expresividad, ya que permite representar multitud de aplicaciones distintas de forma natural al usuario. En esto se diferencia de otros lenguajes: mientras estos lenguajes permiten diseñar multitud de programas dentro de su dominio y sus reglas impuestas, Maude permite al usuario especificar sus propios dominios y trabajar con ellos, sin perderse en formalismos demasiado complicados ni renunciar a que las implementaciones sean eficientes.

Este cambio de enfoque va a resultar clave en cómo representar el problema. Lo primero que haremos será definir las jerarquía de tipos y operadores que gobiernan el comportamiento de las fórmulas lógicas.

Otras implementaciones representan estas características haciendo uso de las estructuras ya existentes, por ejemplo representando el conjunto de cláusulas como listas de listas, que se recorren a partir de bucles `for` o `while`. De esta manera, se pierde parte de la especificación a la hora de traducir a código las estructuras, mientras que en Maude somos capaces de mantener la especificación formal, haciendo más entendible cómo se está abordando el problema.

Los programas en Maude se expresan como teorías de reescritura, tal y como analizamos en la sección 1.2. Los programas se estructuran en módulos, que pueden ser de tres tipos:

- Módulos funcionales: los módulos funcionales nos permiten representar teorías ecuacionales, en las que se incluyen las declaraciones de tipos y subtipos, operadores, variables, ecuaciones y relaciones de pertenencia. Estudiaremos más en detalle cómo funcionan los módulos funcionales en la sección A.2.

- Módulos de sistema: los módulos de sistema representan teorías de reescritura, es decir, se incluyen todo aquello que podemos representar en un módulo funcional y también declaraciones de reglas. Las reglas se entienden como un conjunto de transiciones locales no deterministas que se aplican sobre términos del sistema. Estudiaremos cómo funcionan las reglas más en detalle en la sección A.4.

- Módulos de estrategias: los módulos de estrategias se encuadran dentro del lenguaje de estrategias de Maude. Este lenguaje proporciona métodos para controlar la aplicación de reglas, que como hemos dicho anteriormente, es un proceso totalmente no determinista. Las funcionalidades de este lenguaje venían ya provistas en el nivel de metalenguaje de Maude, pero resulta tedioso y poco entendible para usuarios que se inician en Maude.

  En este caso, hacemos una distinción explícita entre lenguaje de estrategias y módulos de estrategias, pues las estrategias no se tienen que aplicar necesariamente en módulos, sino que se pueden aplicar utilizando ciertos comandos.

  El lenguaje de estrategias no solo se limita al orden en el que se aplican las reglas, sino que también se puede utilizar para expresar situaciones más complejas. Por ejemplo, nos puede servir para poder aplicar reglas paramétricas que necesitan ser instanciadas para poder aplicarse.

  La novedad de este proyecto es estudiar la aplicación de estrategias para la resolución del problema del SAT, por lo que las estudiaremos en profundidad en la sección A.5.

1.4. Contribuciones

El mundo de los resolutores SAT es un mundo muy extenso, donde cada nueva aportación puede estar fundamentada en aspectos muy concretos de la computación, como pueden ser la introducción de nuevas heurísticas que aplicar u optimizaciones sobre el acceso a memoria. En la mayoría de casos, estas descripciones se hacen de manera muy informal, sin detallar los fundamentos teóricos que subyacen, simplemente presentando los resultados.

Tinelli hace alusión a ello en su propio trabajo, en el cuál hace énfasis en el salto que existe entre los fundamentos teóricos y las implementaciones actuales, haciendo difícil entender el porqué de este comportamiento para la gente que no está familiarizada con los conceptos.

Es por ello que su trabajo vincula ambos enfoques, y sienta las bases teóricas que nos permiten entender a un nivel más abstracto cómo funciona. Nosotros compartimos esta filosofía y pretendemos

llevarla un paso más allá. Vamos a partir de las reglas de Tinelli e incorporar nuevas modificaciones que nos permitan estudiar otras técnicas relevantes.

Utilizar lógica de reescritura para abordar este tema parece muy acertado. En efecto, una de las características más destacadas de la lógica de reescritura es que no hay apenas diferencia entre el formalismo de lo que queremos representar y su representación final. Por tanto, la propia implementación en lógica de reescritura del problema del SAT nos aporta información sobre los fundamentos que subyacen al problema.

En particular, el marco que nos proporciona Maude es muy útil para representar de forma clara estas técnicas. Su diferenciación en 3 niveles nos permite separar explícitamente reglas de inferencia y heurísticas, que en la mayor parte de los resolutores SAT se encuentran combinados. Así podremos entender perfectamente cómo funcionan las reglas y estudiar cómo aplicarlas según distintas estrategias.

Por ello la explicación de las distintas técnicas y su correspondiente implementación en Maude están entrelazadas a lo largo de toda la memoria. Este enfoque ha motivado el siguiente esquema en la memoria:

- En el capítulo 2, introduciremos la notación que vamos a emplear en el resto del documento, así como el sistema de reglas DPLL clásico. Reflejaremos estas definiciones y reglas en una serie de módulos, dándonos finalmente la primera estrategia que vamos a probar: `classic-dpll-strat`.

- En el capítulo 3 vamos a estudiar las ecuaciones propuestas por Tinelli en su trabajo [9]. Para ello, también estudiaremos cómo se obtienen las cláusulas de *backjump* en general, y en particular cómo se obtiene aquella que contiene al *first Unique Implication Point*. El capítulo culminará con la segunda estrategia a analizar: `basic-dpll-strat`.

- Siguiendo este hilo, el capítulo 4 se va a centrar en la técnica de *watch-literal*, proponiendo un sistema de reglas alternativo al de Tinelli. Este será el último sistema de reglas que propongamos, y resultará en la tercera estrategia a analizar: `watch-literal-dpll-strat`.

- El capítulo 5 se centrará en algunas de las heurísticas más famosas que se basan en la puntuación de las variables y que sirven para decidir las asignaciones de variable. Estudiaremos una heurística estática, *Jeroslow-Wang*, y una heurística dinámica, *Variable State Independent Decaying Sum*. Estas heurísticas se verán reflejadas en los módulos de estrategias `jw-heuristic-strat` y `vsids-heuristic-strat`.

- Por último, implementaremos un resolutor al completo: *Berkmin*. Este resolutor tiene su propia heurística, y además incluye técnicas para eliminar cláusulas repetidas basadas en la edad de las variables, y reinicios selectivos. Así, explicaremos el último módulo de estrategias que hemos desarrollado: `berkmin-heuristic-strat`. El capítulo 6 pretende servir de esqueleto para futuras adaptaciones de otros resolutores de SAT, y contiene la estructura que a nuestro juicio es menos tediosa y más práctica para llevarlo a cabo.

- Dedicaremos el capítulo 7 a comparar las 6 estrategias que hemos generado con ejemplos aleatorios de 3-SAT. También aprovecharemos para explicar brevemente los fundamentos de generación de estos experimentos, así como una guía para realizar grandes experimentos con Maude.

- En el capítulo 8, recogeremos las ideas principales del trabajo y las conclusiones del mismo.

- Estudiaremos en detalle las funcionalidades y la sintaxis de Maude en el apéndice A. En particular, nos centraremos en discutir aquellas funcionalidades que necesitamos para entender el resto del documento, haciendo especial énfasis en el lenguaje de estrategias. Este lenguaje ha sido incorporado de manera oficial recientemente en la versión 3.0 de Maude, por lo que es menos conocido. Nuestra misión consistirá en poner en valor las múltiples posibilidades que nos abre este nivel.

  En caso de no tener nociones previas sobre el funcionamiento del lenguaje Maude, recomendamos leer este apéndice antes de proceder con el resto de capítulos.

Por último, queremos que este trabajo sirva de guía para futuras implementaciones de sistemas en Maude, ayudando a posibles usuarios a entender más en detalle cómo sacar el máximo partido a este sistema, en particular al lenguaje de estrategias. Todo el código utilizado se incluye en el repositorio https://github.com/alexcere/SAT-Maude.

Existen otros trabajos previos que abordan este tema, mostrando el gran poder expresivo de Maude. Una recopilación se puede encontrar en la web [3].

1.5. Plan de trabajo

El proyecto se ha realizado entre los meses de febrero y julio de 2020. El trabajo se ha repartido en tres grandes bloques:

1. La primera fase del trabajo ha consistido en buscar bibliografía sobre las distintas reglas y heurísticas del problema SAT, así como familiarizarse con el lenguaje Maude. La curva de aprendizaje de este lenguaje es muy pronunciada, así que todo este proceso se ha llevado a cabo a lo largo de los meses de febrero, marzo e inicios de abril.

2. La segunda fase del trabajo ha consistido en adaptar todo el conocimiento de resolutores SAT recopilado en la fase anterior al lenguaje Maude. Se han generado un total de 18 archivos, que contienen el código necesario para llevar a cabo 6 estrategias distintas. Según se iban desarrollando las estrategias, se iban ejecutando los experimentos con las mismas. Este proceso ha abarcado los meses de abril, mayo y parte de junio.

3. Por último, la memoria de este trabajo se ha desarrollado entre mediados de junio y principios de julio.

Capítulo 2

Algoritmo DPLL clásico

En esta sección vamos a introducir la terminología que utilizaremos para discutir el problema del SAT, así como su correspondiente representación en Maude. Con esta terminología, podremos introducir el algoritmo DPLL clásico, y posteriormente ver cómo aplicar sus reglas de forma razonada para sacar el máximo rendimiento.

2.1. Definiciones básicas

Diremos que P es el conjunto de símbolos proposicionales con los que podemos definir una fórmula. En Maude, asumiremos que $P =$ `QID`, es decir, cualquier identificador válido podrá ser un símbolo proposicional. Cualquier elemento $p \in P$ se denotará como variable, de tal manera que una variable tiene dos literales asociados: p y $\neg p$, representando las dos posibles asignaciones de p. La negación de un literal l corresponde a $\neg p$ si l es p, o a p si l es $\neg p$. Diremos que un literal l es un literal de decisión si es de la forma $d(l)$. Estos literales marcan las decisiones que se van tomando a lo largo del algoritmo DPLL, de tal forma que si una búsqueda es infructuosa podemos deshacer la última decisión tomada.

Una cláusula C es una disyunción de literales $C = l_1 \vee ... \vee l_n$, y un conjunto de cláusulas o fórmula F corresponde a la conjunción de una o más cláusulas $F = C_1 \wedge ... \wedge C_n$. La negación de una cláusula corresponde a la conjunción de sus literales negados, es decir, $\neg C = \neg l_1 \wedge ... \wedge \neg l_n$. Al conjunto de cláusulas se representa como $C_1, ..., C_n$. Un literal de decisión no puede formar parte de una cláusula, pues el proceso de decisión no afecta a la forma en la que están construidas las fórmulas.

Una asignación parcial M corresponde a una secuencia de literales, de tal manera que no pueden aparecer un literal y su negación. Un literal es cierto en M si $l \in M$. Un literal está definido en M si $l \in M$ o $\neg l \in M$. Hablaremos de asignación de literales a cierto o falso en aquellos casos en los que añadamos un literal l o su negación $\neg l$ respectivamente a la asignación parcial M, asumiendo que l no está definido en M.

Diremos que M satisface una cláusula C si la cláusula C se cumple dada esa asignación. Además, por comodidad diremos que C es una cláusula de conflicto de M, o $M \models C = \mathit{false}$, si se cumple que $M \models \neg C$.

Diremos que una asignación parcial satisface un conjunto de fórmulas, o equivalentemente $M \models F$, si M satisface cada fórmula $C \in F$. En tal caso, M es un modelo de F. Si no existen modelos de F entonces diremos que F es insatisfactible. Un conjunto de cláusulas F' es consecuencia lógica de otro conjunto F, denotado $F \models F'$, si todas las asignaciones que satisfacen F también satisfacen F'. Si se cumple además que $F' \models F$, entonces diremos que son lógicamente equivalentes.

Llamaremos contexto a una asignación parcial que admite literales de decisión y en la que importa el orden de los elementos. Un contexto representará las decisiones que vayamos tomando a lo largo del proceso.

Dado un contexto concreto, diremos que un literal l pertenece al nivel de decisión x si existen x literales de decisión anteriores a l en ese contexto. En particular, el nivel 0 se corresponde con aquellas variables que se encuentran antes de haber tomado cualquier decisión.

Llamaremos secuencia básica al par de la forma $M \mid\mid F$, siendo M un contexto y F un conjunto de cláusulas. El algoritmo DPLL clásico se basa en un sistema de reglas de inferencia de pares de esta forma, donde la parte F se encuentra fija, y en la parte de contexto, se van a realizar y deshacer las decisiones.

Para ello, se define una relación de transición entre secuencias $\Rightarrow$, de tal manera que una derivación corresponde con una sucesión de secuencias básicas $S_0 \Rightarrow S_1 \Rightarrow ...S_n....$ Un sistema de inferencia se

corresponde a un conjunto de reglas de transición que se aplican sobre una serie de estados. De esta manera, los estados finales corresponden a aquellos estados sobre los que no se puede aplicar ninguna regla de transición.

Todas las definiciones anteriores se encuentran recogidas en el módulo `LOGIC`. Vamos a comentar brevemente aquellas sutilezas que merecen la pena reseñar.

Lo primero que necesitamos es hacer una declaración de tipos que representen los conceptos mencionados anteriormente:

```
sorts Literal Context Clause ClauseSet BasicSequent Sequent .
subsorts Qid < Literal < Context Clause < ClauseSet .
subsort BasicSequent < Sequent .
```

Nótese que esta declaración abstrae algunos de los conceptos anteriores. Por ejemplo, se entiende que una cláusula y un contexto corresponden a agrupaciones de literales, no existiendo una relación directa entre ambos tipos.

Cabe destacar también la definición de una secuencia genérica, que es un supertipo de la secuencia básica mencionada anteriormente. Esto se debe a que vamos a definir distintos sistemas de inferencias, y en uno de los casos no trabajaremos con cláusulas en el sentido que hemos definido anteriormente, sino que utilizaremos cláusulas vigiladas, otro concepto que lo extiende.

A continuación, incluimos las constantes, los constructores y algunas funciones:

```
*** Constantes

op emptyCTX : → Context [ctor] .
op emptyCS : → ClauseSet [ctor] .
op failState : → Sequent [ctor] .
op [] : → Clause [ctor] .

*** Constructores

op ~ : Literal → Literal [ctor] .
op d : Literal → Literal [ctor] .
op __ : Context Context → Context [ctor assoc id: emptyCTX prec 30] .
op _,_ : ClauseSet ClauseSet → ClauseSet [ctor assoc comm id: emptyCS prec 30] .
op _\/_ : Clause Clause → Clause [ctor assoc comm id: ([]) prec 20] .
op _||_ : Context ClauseSet → BasicSequent [ctor] .

*** Operadores

op _in_ : Literal Context → Bool .
op _|=_ : Context Clause → [Bool] .
op definedLiteral : Literal Context → Bool .
op existsDecisionLiteral : Context → Bool .
```

Hemos incluido un elemento neutro por cada uno de nuestros tipos iniciales. En la mayoría de casos, la elección ha sido para facilitar la declaración de funciones con encaje de patrones. Sin embargo, un contexto vacío representa aquella toma de decisiones en la que aún no hay ninguna decisión tomada, que se corresponde con el contexto inicial.

El uso de atributos es extremadamente importante para lograr nuestro propósito. En el caso del operador `__`, hemos evitado incluir el atributo `comm` para que no se modifique el orden en el que tomamos las decisiones. Por otro lado, en el operador `_\/_` hemos hecho justo lo contrario, pues no nos importa el orden entre literales dentro de una cláusula.

Las precedencias entre operadores están incluidas de tal forma que primero se agrupe por cláusulas, y luego por conjuntos de cláusulas. De esta forma, la expresión sin paréntesis de una secuencia básica puede ser parseada sin problemas.

El resto de declaraciones corresponden a ecuaciones que nos ayudarán posteriormente a definir otras ecuaciones o reglas. Vamos a destacar las ecuaciones más relevantes para definir el sistema de reglas:

```
eq ~(~(l)) = l .
eq l \/ l \/ C = l \/ C .

eq l in M l N = true .
eq l in M d(l) N = true .
```

```
eq l in M = false [owise] .

eq definedLiteral(l, (M l CTX)) = true .
eq definedLiteral(l, (M ~(l) CTX)) = true .
eq definedLiteral(~(l), (M l CTX)) = true .
eq definedLiteral(l, (M d(~(l)) CTX)) = true .
eq definedLiteral(~(l), (M d(l) CTX)) = true .
eq definedLiteral(l, (M d(l) CTX)) = true .
eq definedLiteral(l, M) = false [owise] .

eq CTX |= [] = false .
eq M l N |= l \/ C = true .
eq M d(l) N |= l \/ C = true .
eq M ~(l) N |= l \/ C = M ~(l) N |= C .
eq M l N |= ~(l) \/ C = M l N |= C .
eq M d(~(l)) N |= l \/ C = M d(~(l)) N |= C .
eq M d(l) N |= ~(l) \/ C = M d(l) N |= C .

eq existsDecisionLiteral(M d(l) N) = true .
eq existsDecisionLiteral(M) = false [owise] .
```

Aunque exista una ecuación que simplifica la doble negación, en aquellas ecuaciones en las que trabajemos con un literal y su negación mediante encaje de patrones necesitamos incluir explícitamente una ecuación por cada combinación.

En el caso del operador `_|=_`, no hemos incluido una ecuación con el atributo `owise`. Esto se debe a que tal y como lo hemos definido anteriormente, el resultado de aplicarlo sobre una expresión puede tener 3 resultados distintos: true si la cláusula se cumple con el contexto actual, false si la negación de la cláusula se cumple, e indefinido si no se da ninguna de las situaciones anteriores. En este caso, conseguimos este comportamiento gracias al *kind*, que hace de representación de todos aquellos términos que no se reducen completamente, y por ende, de las cláusulas que no están satisfechas ni son contradictorias.

2.2. Sistema de reglas de inferencia

En el apartado anterior nos referíamos al símbolo $\Rightarrow$ como la relación de transición, coincidiendo exactamente con el símbolo que se utiliza para definir reglas en Maude. Este hecho no es casualidad, pues precisamente coincide con las transiciones locales que se especifican a nivel de reglas. De nuevo, encontramos otra situación en la que el formalismo natural del problema se adapta al marco de la lógica de escritura de forma directa.

En este apartado, la definición de las reglas se hará directamente sobre Maude. Hemos decidido proceder así para ilustrar claramente cómo funcionan estas reglas, y cómo se apoyan en los operadores que hemos definido previamente. También hemos decidido mantener los nombres originales en inglés, pues así el lector tiene una idea directa de la terminología en caso de buscar más referencias.

El sistema de inferencia clásico se encuentra recogido en el módulo `CLASSIC-DPLL` y consta de las siguientes reglas:

1. `UnitPropagate` : si nos encontramos alguna cláusula en la que todos los literales son falsos salvo uno que está sin definir aún, entonces necesariamente tenemos que asignar ese literal a cierto.

   Esta regla es fundamental dentro del esquema CDCL, pues los resolutores actuales pueden llegar a ocupar entre el 80 % o 90 % buscando cláusulas sobre las que aplicar esta regla.

   ```
   crl [UnitPropagate] : M || (l \/ C), F ⇒ M l || (l \/ C), F
     if M |= C = false ∧ definedLiteral(l, M) = false .
   ```

2. `PureLiteral` : esta regla se utiliza para asignar a cierto directamente aquellos literales cuya negación no aparece en ninguna otra cláusula del conjunto.

   ```
   crl [PureLiteral] : M || (l \/ C), F ⇒ M l || (l \/ C), F
     if (~(l) in F) = false ∧ definedLiteral(l,M) = false .
   ```

3. `Decide` : esta regla nos permite generar las distintas combinaciones posibles de decisiones. Cada vez que aplicamos esta regla, marcamos el literal que hemos decidido como un literal de decisión.

$\varnothing$	$\|\|$	F	$\Rightarrow$	`PureLiteral`
10	$\|\|$	F	$\Rightarrow$	`Decide`
10 $d(1)$	$\|\|$	F	$\Rightarrow$	`UnitPropagate`, $C = \neg 1 \vee 2$
10 $d(1)$ 2	$\|\|$	F	$\Rightarrow$	`Decide`,
10 $d(1)$ 2 $d(3)$	$\|\|$	F	$\Rightarrow$	`UnitPropagate`, $C = \neg 3 \vee 4$
10 $d(1)$ 2 $d(3)$ 4	$\|\|$	F	$\Rightarrow$	`Decide`
10 $d(1)$ 2 $d(3)$ 4 $d(5)$	$\|\|$	F	$\Rightarrow$	`UnitPropagate`, $C = \neg 5 \vee 6$
10 $d(1)$ 2 $d(3)$ 4 $d(5)$ 6	$\|\|$	F	$\Rightarrow$	`Decide`
10 $d(1)$ 2 $d(3)$ 4 $d(5)$ 6 $d(7)$	$\|\|$	F	$\Rightarrow$	`UnitPropagate`, $C = \neg 7 \vee 8$
10 $d(1)$ 2 $d(3)$ 4 $d(5)$ 6 $d(7)8$	$\|\|$	F	$\Rightarrow$	`UnitPropagate`, $C = \neg 1 \vee \neg 8 \vee 9$
10 $d(1)$ 2 $d(3)$ 4 $d(5)$ 6 $d(7)$ 8 9	$\|\|$	F	$\Rightarrow$	`Backtrack`, $C = \neg 4 \vee \neg 7 \vee \neg 9$
10 $d(1)$ 2 $d(3)$ 4 $d(5)$ 6 $\neg 7$	$\|\|$	F	$\Rightarrow$	$\ldots$

Figura 2.1: Derivación utilizando las reglas del algoritmo DPLL clásico

Nótese que solo podemos tomar decisiones sobre variables que no están incluidas aún en el contexto actual, y cuyo literal asociado aparece en alguna cláusula de la fórmula.

```
crl [Decide] : M || (l \/ C),F ⇒ M d(l) || (l \/ C),F
  if definedLiteral(l,M) = false .
```

4. `Fail` : la regla `Fail` nos permite concluir que hemos llegado a que no existe ninguna asignación que satisfaga el problema. Es decir, el problema es insatisfactible.

   Para poder aplicar esta regla y generar un estado de error, es necesario que nuestro contexto actual no contenga ningún literal de decisión. En tal caso, podríamos tomar su literal negado y seguir explorando en busca de asignaciones válidas.

   ```
   crl [Fail] : M || F,C ⇒ failState if M |= C = false
     ∧ existsDecisionLiteral(M) = false .
   ```

5. `Backtrack` : la regla de vuelta atrás se aplica cuando tenemos una cláusula de conflicto y hay algún literal de decisión en el contexto actual.

   En este caso, elegimos el literal de decisión más reciente, y lo reemplazamos por su negación. En este caso, no volvemos a anotarlo como literal de decisión, pues ya sabemos que la otra posibilidad ha sido explorada ya.

   ```
   crl [Backtrack] : M d(l) N || F,C ⇒ M ~(l) || F,C if
     M d(l) N |= C = false ∧ existsDecisionLiteral(N) =/= true .
   ```

Veamos con un ejemplo cómo se aplican estas reglas.

Ejemplo 2.1. *En la figura 2.1, se ve cómo hemos aplicado las distintas reglas del algoritmo DPLL, tomando como fórmula F:*

$$F = \neg 1 \vee 2,\ \neg 3 \vee 4,\ \neg 5 \vee 6,\ \neg 7 \vee 8,\ \neg 1 \vee \neg 8 \vee 9,\ \neg 4 \vee \neg 7 \vee \neg 9, 1 \vee \neg 4 \vee \neg 6 \vee 10,\ 3 \vee \neg 4 \vee 5 \vee 10$$

2.3. DPLL clásico a nivel de estrategias

El sistema de reglas del apartado anterior es completo, por lo que podríamos intentar aplicarlas sin tener en cuenta el orden, y finalmente llegaríamos a una asignación válida, o a deducir que el problema es insatisfactible.

Sin embargo, existen estrategias que minimizan el número de reglas a aplicar. Vamos a comentar una estrategia general, basada en parte en el esquema CDCL, pero que va a ser común a todas las otras estrategias que comentemos. Para ello, vamos a reflexionar sobre las reglas del apartado anterior.

La regla `PureLiteral` asigna aquellos literales cuya negación no aparece en ninguna otra cláusula, por lo que podemos considerar la aplicación de esta regla como un paso previo de preprocesado. Además, podríamos también eliminar las cláusulas en las que aparecen estos literales, pues desde el punto de vista

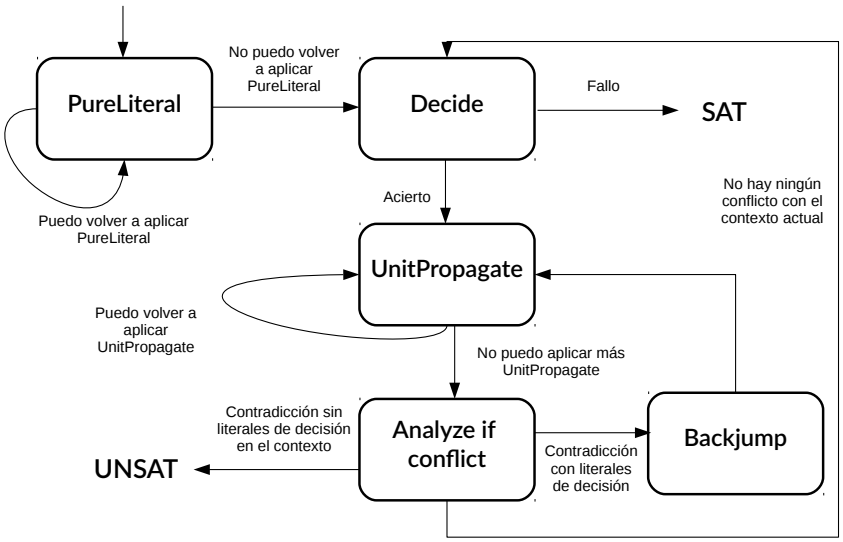


Figura 2.2: Estrategia para el algoritmo DPLL clásico

de deducir o no la satisfactibilidad de la fórmula, no nos van a aportar información. Esto se debe a que están directamente resueltas. En futuros sistemas de inferencia, adoptaremos este comportamiento.

A nivel de Maude, aplicar la regla `PureLiteral` hasta que no se puede aplicar más corresponde a la estrategia

```
PureLiteral !
```

Ahora sigamos con el resto de reglas. La regla `Decide` nos permite probar asignaciones, de tal manera que podemos deshacerlas con la regla `Backtrack` en caso de no alcanzar ninguna asignación válida. Es coherente intentar aplicar lo menos posible esta regla, pues cuantas más decisiones hagamos, más decisiones nos tocará deshacer. Es por ello que la regla `UnitPropagate` es la que la mayoría de resolutores pasan mayor tiempo utilizando.

A nivel de estrategias, esto se traduce en intentar aplicar la regla `UnitPropagate` tantas veces como se pueda antes de tomar una decisión. En este punto, podríamos seguir dos alternativas. La primera consistiría en comprobar si he llegado a una cláusula de conflicto cada vez que añado un literal con `UnitPropagate`. Esta alternativa es costosa, pues implica recorrer todas las cláusulas comprobando si efectivamente se ha llegado a tal situación.

Por ello tiene más sentido hacer la comprobación de error una vez no se pueda aplicar más la regla `UnitPropagate`. En este punto, comprobamos si puedo aplicar la regla `Backtrack`. En caso de poder aplicarla, vuelvo a intentar aplicar `UnitPropagate`, ya que el nuevo literal asignado me puede disparar nuevas aplicaciones de esta regla.

En caso contrario, intento aplicar la regla `Fail`. Si no se verifica, entonces vuelvo a tomar una nueva decisión y se repite todo el proceso.

El proceso se acaba una vez no pueda decidir más literales. En ese momento, al no haber generado ninguna contradicción con el contexto actual, directamente tengo que se satisfacen todas las cláusulas de mi conjunto.

Todo este razonamiento se recoge en el esquema mostrado en la figura 2.2. Cabe destacar que todas las estrategias propuestas en esta memoria siguen la misma filosofía, y difieren básicamente en la forma de actualizar o generar información.

El ejemplo 2.1 sigue precisamente este esquema a la hora de aplicar las reglas.

También cabe reseñar que siguiendo este proceso hacemos comprobaciones repetidas. Esto es evidente cuando terminamos de aplicar la regla `UnitPropagate` e intentamos aplicar las reglas `Fail` y `Backtrack` de forma seguida, comprobando en ambos casos si hay alguna cláusula que no se satisface. Las reglas de inferencia de las sección 3.1 no tienen en cuenta este comportamiento, pero el sistema de reglas de inferencia de la sección 4.2 estará específicamente diseñado para evitar estas comparaciones repetidas. Esto implicará una pérdida de la completitud del sistema de reglas, lo que obligará a que necesariamente haya que controlar su ejecución desde el nivel de estrategias.

Las estrategias del módulo `CLASSIC-DPLL-STRATEGY` son las siguientes:

```
sd basic-iteration := UnitPropagate ! ; Backtrack ?
  basic-iteration : (Fail or-else success-branch) .

sd success-branch := one(Decide) ? basic-iteration : idle .

sd classic-dpll-strat := PureLiteral ! ; basic-iteration .
```

Lo último que nos queda por discutir es qué comando utilizar para ejecutar la estrategia `classic-dpll-strat`. El sistema de reglas de inferencia es completo, por lo que no necesitamos explorar distintas ramas del árbol de ejecución. Además, estaremos interesados en obtener únicamente una asignación válida. Por tanto, vamos a ejecutar esta estrategia utilizando el comando `dsrew [1]`. Podemos ver su uso con el siguiente ejemplo:

```
Maude> dsrew [1] in CLASSIC-DPLL-STRATEGY : emptyCTX || '1 \/ ~('2),
  ~('1) \/ '2 \/ ~('3), '3 \/ '2 using classic-dpll-strat .

...

Solution 1
rewrites: 109 in 0ms cpu (0ms real) (~ rewrites/second)
result BasicSequent: d('1) '2 d('3) || '1 \/ ~('2),'2 \/ '3,'2 \/ ~('1) \/ ~( '3)
```

Utilizaremos este mismo comando para el resto de apartados.

Capítulo 3

Algoritmo DPLL con aprendizaje

El algoritmo DPLL clásico se vuelve extremadamente costoso según vamos aumentando el número de literales que aparecen en las cláusulas a considerar. Es por ello que la mayoría de resolutores SAT actuales no consideran directamente este sistema de reglas. Una forma de mejorar este algoritmo es a través de mecanismos de vuelta atrás más potentes, que permitan deshacer más de una decisión a la vez.

La idea general de estos mecanismos se basa en analizar las causas que han motivado la aparición de un conflicto. Podríamos asumir que un conflicto se debe exclusivamente a la última decisión que hemos tomado, que es al fin y al cabo el razonamiento detrás de la regla `Backtrack`. Sin embargo, existen casos en los que el conflicto venía generado por decisiones previas, por lo que podemos retroceder a decisiones anteriores para explorar el árbol, evitando explorar ramas innecesarias.

El algoritmo DPLL propuesto por Tinelli está basado en este planteamiento y lo estudiaremos más en detalle en la sección 3.1. Este sistema aisla el proceso de obtención de cláusulas que permitan deshacer varias decisiones a la vez de las reglas propiamente dichas.

Veamos con un ejemplo cómo podemos tener conflictos cuyo origen viene dado por decisiones anteriores a la última.

Ejemplo 3.1. *Siguiendo el ejemplo 2.1, veamos cómo podríamos haber deshecho varias decisiones de una vez en lugar de aplicar la regla de* `Backtrack`*.*

Vamos a considerar la cláusula $\neg 1 \vee \neg 4 \vee \neg 7$*. Esta cláusula cumple que* $F \models \neg 1 \vee \neg 4 \vee \neg 7$*, como probaremos en el ejemplo 3.2. Si consideramos el contexto* $10\ d(1)\ 2\ d(3)\ 4\ d(5)\ 6\ d(7)\ 8\ 9$*, es claro que la cláusula anterior contradice al mismo. Además, se deduce que si* 1 *y* 4 *están en un contexto, entonces necesariamente tenemos que* $\neg 7$*. Por tanto, en vez de deshacer la decisión* 7*, vemos que podemos también deshacer la decisión* 5*, teniendo así el contexto final* $10\ d(1)\ 2\ d(3)\ 4\ \neg 7$*.*

En la sección 3.2 estudiaremos en más detalle cómo generar cláusulas con las que deshacer una o varias decisiones, a las que llamaremos cláusulas de backjump.

Una cláusula de backjump es de la forma $C' \vee l'$, de tal forma que si la hubiésemos considerado en niveles de decisión anteriores, entonces hubiésemos aplicado la regla `UnitPropagate` a l'. En el ejemplo anterior, tenemos que $C' = \neg 1 \vee \neg 4$ y $l' = \neg 7$.

Para generar una cláusula de backjump válida, se utiliza la regla de resolución binaria, que consiste en la siguiente derivación:

$$\frac{a_1 \vee a_2 \vee ... \vee a_n \vee c \qquad b_1 \vee b_2 \vee ... \vee b_m \vee \neg c}{a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_m}$$

Veamos cómo aplicar esta regla con un ejemplo:

Ejemplo 3.2. *En el ejemplo 3.1, estudiamos la cláusula de backjump* $\neg 1 \vee \neg 4 \vee \neg 7$*. Veamos ahora que esta cláusula se puede obtener a partir de la fórmula inicial aplicando la regla de resolución binaria. Para ello, expresaremos las distintas aplicaciones de la regla como un árbol de derivación, partiendo de la cláusula de conflicto como cláusula inicial:*

$$\dfrac{\dfrac{\neg 4 \vee \neg 7 \vee \neg \mathbf{9} \qquad \neg 1 \vee \neg 8 \vee \mathbf{9}}{\neg 1 \vee \neg 4 \vee \neg 7 \vee \neg \mathbf{8}} \qquad \neg 7 \vee \mathbf{8}}{\neg 1 \vee \neg 4 \vee \neg 7}$$

Nótese que las cláusulas que hemos utilizado coinciden con las cláusulas que hemos ido utilizando para aplicar la regla `UnitPropagate` *en orden inverso de propagación. En general, podremos generar cláusulas de backjump utilizando este procedimiento, como estudiaremos en la sección 3.2.*

Esta regla nos permite generar cláusulas de tal forma que sean consecuencia lógica de las de partida. Por tanto, cuando llegamos a un conflicto, podemos utilizar cualquier nueva cláusula que sea conflictiva con el contexto actual generada utilizando esta regla como cláusula de backjump, en lugar de utilizar la cláusula de conflicto.

En teoría, podríamos probar a generar cláusulas a partir de la aplicación arbitraria de la regla de resolución binaria hasta dar con una cláusula contradictoria con el contexto actual que nos sirva como cláusula de backjump. En la práctica, es inasumible para un resolutor SAT, pues en la mayoría de casos no obtendríamos cláusulas útiles.

Por ello vamos a aprovechar la información de un conflicto para generar esta cláusula, pues existen algoritmos que garantizan la obtención de cláusulas de backjump de forma eficiente. Este planteamiento motiva el esquema de aprendizaje de cláusulas guiado por conflictos (CDCL), que ha probado ser muy eficiente y es la base de todos los resolutores modernos. Estudiaremos este esquema en la sección 3.3, que junto a las ideas propuestas en la sección 3.2, nos va a servir de base para proponer estrategias eficientes.

3.1. Sistema de reglas de inferencia

En esta sección, vamos a comentar los distintos sistemas de reglas en SAT propuestos en el trabajo de Tinelli [9].

Lo primero que vamos a estudiar es un sistema de reglas completo que permite deshacer varias decisiones de una vez. A este primer sistema lo denotaremos como algoritmo DPLL básico. Este algoritmo tiene las reglas `UnitPropagate`, `Decide` y `Fail` definidas en la sección 2.2. Además, consta de una regla extra que sustituye a la regla `Backtrack`, que denotaremos como `Backjump`. Desde este punto de la memoria, nos referiremos al proceso de Backjump como la aplicación de esta regla.

La regla `Backjump` simula el proceso de deshacer varias decisiones de una vez. Para ello, utiliza una cláusula de backjump en el sentido que hemos definido anteriormente, de tal manera que la regla nos quedaría

```
crl [Backjump] : M d(x) N || CS, C ⇒ M l || CS,C if (M d(x) N) |= C = false ∧
  CS,C |= C' ∧ C'' := removeLiteralFromClause(l, C') ∧ M |= C'' = false ∧
  M |= C'' =/= false ∧ definedLiteral(l, M) = false ∧ definedLiteral(l, (CS,C)) .
```

donde C' es la cláusula de backjump, y es de la forma $C' = C'' \vee l$.

Esta regla no la hemos implementado directamente en Maude, pues si la consideramos tal cual a nivel de reglas en Maude, nos daríamos cuenta de que la aparición de la cláusula de backjump en las condiciones hacen que esta regla sea directamente no admisible.

Siendo puristas, podríamos haber definido esta regla con el atributo `nonexec` y definiendo un operador auxiliar que representase la consecuencia lógica. Sin embargo, como el proceso de generación de la cláusula de backjump siempre va a ser independiente de la aplicación de las reglas, hemos decidido asumir que la cláusula de backjump es consecuencia lógica de la fórmula inicial y que el literal a propagar aparece en alguna cláusula de la fórmula.

También hemos introducido otro cambio con respecto a esta regla pues a nivel de aplicación su formulación no es idónea, al ser no determinista. Este comportamiento lo podemos ver en el siguiente ejemplo:

Ejemplo 3.3. *Retomamos la secuencia básica y la cláusula de backjump del ejemplo 3.1. En este caso, podemos aplicar la regla* `Backjump` *de dos formas distintas:*

$$\begin{array}{lllll} 10\ d(1)\ 2\ d(3)\ 4\ d(5)\ 6\ d(7)\ 8\ 9 & || & F & \Rightarrow & 10\ d(1)\ 2\ d(3)\ 4\ d(5)\ 6\ \neg 7 \quad || \quad F \\ 10\ d(1)\ 2\ d(3)\ 4\ d(5)\ 6\ d(7)\ 8\ 9 & || & F & \Rightarrow & 10\ d(1)\ 2\ d(3)\ 4 \neg 7 \quad || \quad F \end{array}$$

puesto que se cumple tanto

$$10\ d(1)\ 2\ d(3)\ 4\ d(5)\ 6\ d(7) \models \neg 1 \vee \neg 4 = \mathit{false}$$

como

$$10\ d(1)\ 2\ d(3)\ 4\ d(5) \models \neg 1 \vee \neg 4 = \mathit{false}$$

Está claro que estamos interesados solo en la segunda aplicación, pues es la que retrocede a un nivel inferior.

El problema con la regla `Backjump` es que no identifica por sí misma cuál es el menor nivel al que puede retroceder. Por ello, hemos decidido separar esta regla en dos reglas distintas:

```
crl [Backjump1] : M d(x) N d(y) CTX || CS, C ⇒ M d(x) N l || CS,C
  if (M d(x) N d(y) CTX) |= C == false ∧ existsDecisionLiteral(N) =/= true ∧
  C'' := removeLiteralFromClause(l, C') ∧ M d(x) N |= C'' == false ∧ M |= C'' =/= false ∧
  definedLiteral(l, M d(x) N) =/= true [nonexec] .
crl [Backjump2] : M d(x) N || C,CS ⇒ M l || C,CS if M d(x) N |= C == false ∧
  existsDecisionLiteral(M) =/= true ∧ C'' := removeLiteralFromClause(l, C') ∧
  M |= C'' == false ∧ definedLiteral(l, M) =/= true [nonexec] .
```

La primera regla representa la posibilidad de que los niveles anteriores al nivel correspondiente a `x` no contradicen la cláusula de backjump, pero considerando este nivel sí llegamos a contradicción. Por tanto, hay que deshacer las decisiones posteriores al nivel `x`, que vienen identificadas a partir de la siguiente decisión: `y`. Expresamos que el nivel de `y` es inmediatamente posterior al de `x` a partir de la ecuación `existsDecisionLiteral(N) =/= true`.

Esta regla por sí misma no cubre todos los casos, pues podría pasar que el salto se produce al nivel 0 del contexto actual. Para ello, hemos incluido también la segunda regla. Estas reglas son excluyentes entre sí, pues en este caso, nos aseguramos que el nivel 0 contradice la cláusula de backjump a partir de las condiciones `existsDecisionLiteral(N) =/= true` y `M |= C''`.

Nótese que la regla `PureLiteral` ha quedado fuera del sistema de reglas, pues se entiende que podemos hacer una fase de preprocesamiento en la que aplicamos repetidamente esta regla.

Este sistema de reglas se puede ampliar para que permita otras funcionalidades básicas de otros resolutores SAT: el aprendizaje de cláusulas, la eliminación de las mismas y los reinicios selectivos. A este sistema de reglas lo denotaremos como algoritmo DPLL con aprendizaje, que extiende al algoritmo DPLL básico introduciendo las siguientes reglas:

```
crl [Learn] : M || CS ⇒ M || CS,C if CS |= C [nonexec] .
crl [Forget] : M || CS, C ⇒ M || CS if CS |= (C,CS) .
rl [Restart] : M || CS ⇒ emptyCTX || CS .
```

Las reglas `Learn` y `Forget` representan la posibilidad de añadir y eliminar cláusulas a la fórmula actual, de tal forma que tengamos más cláusulas sobre las que razonar. Nos podríamos preguntar por qué la regla `Forget` es necesaria. En la implementación de resolutores, no es asumible mantener una fórmula excesivamente larga, pues aplicaciones de reglas como `UnitPropagate` se volverían demasiado ineficientes. Por ello esta regla tiene interés a nivel práctico.

La misma pregunta nos podía ocurrir con la regla `Restart`. Esta regla deshace todas las decisiones que hemos tomado, lo que nos podría resultar chocante a priori. Sin embargo, esta regla sí tiene sentido combinada con la regla `Learn`. Esto se debe a que en el momento de reinicio tenemos más información que al iniciar la exploración, por lo que podemos tomar decisiones más informadas y tener la posibilidad de abandonar caminos que podrían ser infructuosos.

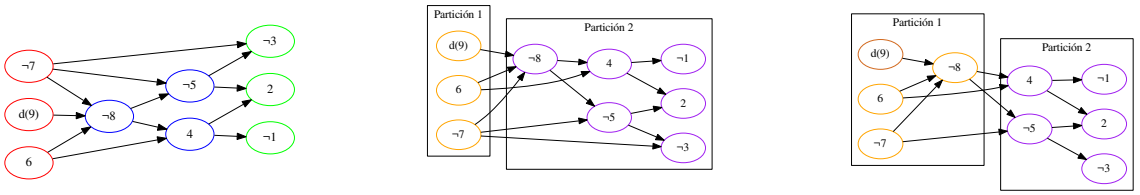
Para no perder completitud con esta regla, es necesario que su aplicación sea monótona: es decir, que el número de pasos de derivación entre dos aplicaciones de esta regla sea estrictamente creciente.

En Maude, como no hemos definido el operador `_|=_` para estudiar la consecuencia lógica y como vamos a controlar la ejecución de estas reglas, hemos decidido implementar las reglas `Learn` y `Forget` como reglas incondicionales, teniendo así las siguientes reglas:

```
rl [Learn] : M || CS ⇒ M || CS,C [nonexec] .
rl [Forget] : M || CS, C ⇒ M || CS [nonexec]
rl [Restart] : M || CS ⇒ emptyCTX || CS .
```

Esto se debe a que en la práctica siempre aplicaremos estas reglas de forma coherente con estas condiciones. Por ejemplo, en el caso de la regla `Forget`, solo olvidaremos cláusulas que han sido previamente aprendidas.

Las reglas `Forget` y `Restart` no se van a considerar a nivel de estrategias, pues para aplicarlas de forma eficiente, se suelen combinar con heurísticas particulares. Hemos decidido incluirlas en esta sección para terminar de explicar el sistema de reglas de inferencia propuesto por Tinelli, y que así el lector tenga una visión global sobre estos sistemas de reglas. Igualmente, en el capítulo 6 estudiaremos heurísticas para la aplicación de estas reglas.



(a) Grafo inicial

(b) Corte considerando únicamente las variables de decisión

(c) Corte considerando el UIP

Figura 3.1: Grafo de conflicto del ejemplo 3.4 y posibles cortes

3.2. Generación de cláusulas de backjump

En esta sección vamos a introducir las ideas principales para generar una cláusula de backjump válida a partir de un conflicto. Todas las ecuaciones necesarias para llevarlo a cabo se encuentran en el módulo `UNIQUE-IMPLICATION-POINT`.

Para ello, primero necesitamos introducir el concepto de información de conflicto. La información de conflicto es una representación de la información que hemos utilizado para generar el contexto actual. Por cada literal de ese contexto, almacenamos qué cláusula lo ha propagado en caso de haberse obtenido aplicando la regla `UnitPropagate`, o la cláusula vacía en caso de haber sido consecuencia de una decisión.

En Maude, lo representaremos como una lista de pares que se encuentra ordenado conforme al orden en el que aparecen los literales en el contexto, de tal forma que su declaración es la siguiente

```
sorts ConflictInfo ConflictInfoSet .
subsort ConflictInfo < ConflictInfoSet .

op emptyCIS : → ConflictInfoSet [ctor] .

op _→_ : Literal Clause → ConflictInfo [ctor prec 22] .
op __ : ConflictInfoSet ConflictInfoSet → ConflictInfoSet [ctor assoc id: emptyCIS prec 30]
```

La idea detrás de la información de conflicto es que si nos restringimos al último nivel de aplicación, podemos generar un grafo dirigido en el que existe un método general para generar cláusulas de backjump. A este grafo lo denotaremos como grafo de conflicto. Los nodos vienen dados por literales que aparecen en el contexto, y las aristas relacionan el conjunto de literales que han llevado a la aplicación de cada regla de `UnitPropagate` con el literal deducido de su aplicación.

Una vez tenemos construido este grafo, una forma de construir una cláusula de backjump consiste en cortar el grafo en dos conjuntos, de tal manera que uno de ellos contenga todos los vértices con grado de entrada 0; y el otro, todos los vértices con grado de salida 0. La cláusula de backjump consiste en la conjunción de la negación de aquellos vértices que se encuentran en la frontera del primer conjunto.

Veamos un ejemplo de cómo se ha generado la información de conflicto y el grafo de conflicto a partir de un ejemplo sacado directamente del trabajo de Tinelli [9]:

Ejemplo 3.4. *Supongamos que tenemos el contexto* ... 6 ... ¬7 ... $d(9)$ ¬8 ¬5 4 ¬1 2 ¬3 *, la información de contexto*

$$9 \mapsto [] \; \neg 8 \mapsto (9 \vee \neg 6 \vee 7 \vee \neg 8) \; \neg 5 \mapsto (8 \vee 7 \vee \neg 5) \; 4 \mapsto (\neg 6 \vee 8 \vee 4) \; \neg 1 \mapsto (\neg 4 \vee \neg 1) \; 2 \mapsto (\neg 4 \vee 5 \vee 2) \; \neg 3 \mapsto (5 \vee 7 \vee 3)$$

y la cláusula conflictiva es $1 \vee \neg 2 \vee 3$.

Entonces, podríamos construir el siguiente grafo de contexto de la figura 3.1a. En esta figura, los literales con grado de entrada 0 vienen representados por los nodos de color rojo, los de grado de salida 0 son de color verde, y el resto de color azul.

Ahora vamos a estudiar algunos posibles cortes de este grafo y analizar la cláusula de backjump resultante en cada caso. Para ello, vamos a analizar los cortes de las figuras 3.1b y 3.1c. En estas figuras, hacemos explícita la partición que hemos realizado, representando los nodos amarillos a los nodos de la frontera del primer conjunto, los nodos dorados a los elementos del primer conjunto que no forman parte de la frontera, y los nodos púrpuras a los elementos del segundo conjunto.

En el caso de la figura 3.1b, obtenemos la cláusula $\neg 6 \vee \neg 9 \vee 7$. *Esta cláusula nos haría deshacer la última decisión tomada, pues deducimos que* $6 \wedge 9 \implies \neg 7$.

La cláusula de backjump asociada a la figura 3.1c es $\neg 6 \vee 7 \vee 8$. *En este caso, deducimos que el conflicto venía de decisiones previas al último nivel, pues* $6 \wedge \neg 7 \implies 8$.

Nótese que en el último caso, la cláusula de conflicto es muy distinta a la cláusula de backjump, no llegando a compartir ningún literal. Además, en función del corte que elijamos, podemos generar cláusulas de backjump más o menos útiles. No existe una forma óptima de obtener esta cláusula; es más, los resolutores abordan este planteamiento con distintas técnicas.

La intuición de por qué este razonamiento funciona radica en entender cómo funciona el grafo de conflicto. Los vértices cuyo grado de entrada es 0 representan a aquellos literales que o bien se encuentran en niveles anteriores al actual, o bien corresponden a la última decisión. Los vértices cuyo grado de salida es 0 representan los literales de cuya asignación no se ha deducido ninguna otra asignación, y que por tanto incluyen al literal que ha entrado en conflicto.

Al cortar el conjunto en dos, hemos conseguido agrupar en un conjunto todas las decisiones previas, mientras que en el otro agrupamos todas las posibles contradicciones. Los literales de la frontera del corte representan un conjunto de literales que de ser tomados así, se propagan hasta alcanzar el conjunto de posibles contradicciones. De esto se deduce que al menos uno de estos literales no puede estar en el contexto actual, o equivalentemente, que necesariamente uno de ellos se tiene que encontrar negado. Esto motiva a que la conjunción de los literales negados de la frontera es una cláusula que es consecuencia lógica de las anteriores.

Ahora bien, para que nos sirva de cláusula de backjump, necesitamos que uno y solo uno de los literales de este corte pertenezca al nivel actual, para que así podamos deducir que necesariamente este literal es incompatible con el resto, y poder deshacer las decisiones asociadas al resto de literales. A este literal se le conoce como *Unique Implication Point* (UIP). Estos literales se pueden identificar en el grafo como aquellos que cumplen que todos los caminos del grafo de conflicto que parten del vértice correspondiente a la última decisión los incluyen necesariamente. Este conjunto es siempre no vacío, pues en particular el vértice correspondiente a la última decisión cumple la condición trivialmente.

Un corte válido que incluye este UIP consiste en coger como primer conjunto únicamente todos los vértices de grado de entrada 0. La cláusula de backjump generada siempre retrocede un nivel, por lo que podríamos entender que la regla de `Backtrack` correspondería a este caso.

En general, nos interesa considerar otros UIP distintos para generar la cláusula de backjump. En particular, muchos de los resolutores SAT modernos generan cláusulas de backjump a partir del *first* UIP, que se corresponde al primer UIP en orden reverso de propagación en el grafo. Para encontrar esta cláusula, no es necesario construir explícitamente el grafo, sino que podemos trabajar directamente con la información de conflicto.

El algoritmo es sencillo: partimos de la cláusula de conflicto y aplicamos repetidamente la regla de resolución binaria con la última cláusula propagada (la cláusula asociada al último término de la información de conflicto actual) hasta que obtengamos una cláusula que solo tenga un literal en el nivel actual. No es necesario almacenar explícitamente el nivel de los literales del contexto para llevar a cabo esta comprobación: los literales del último nivel de decisión corresponden a aquellos literales que aparecen como claves en el mapa. Su declaración en Maude es la siguiente:

```
op literalsInLevel : ConflictInfoSet Clause → Nat .

eq literalsInLevel(CIS (l → C) Cis , l \/ D) = 1 + literalsInLevel(CIS Cis, D) .
eq literalsInLevel(CIS (l → C) Cis , ~(l) \/ D) = 1 + literalsInLevel(CIS Cis, D) .
eq literalsInLevel(CIS (~(l) → C) Cis , l \/ D) = 1 + literalsInLevel(CIS Cis, D) .
eq literalsInLevel(CIS, C) = 0 [owise] .


op obtainBackjumpClause : ConflictInfoSet Clause → Clause .

ceq obtainBackjumpClause(CIS, C) = C if literalsInLevel(CIS,C) == 1 .
ceq obtainBackjumpClause(CIS l → (C \/ l) , ~(l) \/ D) = obtainBackjumpClause(CIS, C \/ D)
  if literalsInLevel(CIS l → (C \/ l), ~(l) \/ D) =/= 1 .
ceq obtainBackjumpClause(CIS ~(l) → (C \/ ~(l)),l \/ D) = obtainBackjumpClause(CIS, C \/ D)
  if literalsInLevel(CIS ~(l) → (C \/ ~(l)), l \/ D) =/= 1 .
eq obtainBackjumpClause(CIS l → C, D) = obtainBackjumpClause(CIS, D) [owise] .
```

Como hemos indicado anteriormente, `literalsInLevel` obtiene el número de literales del nivel de decisión actual contando el número de ocurrencias de literales en la cláusula actual que aparecen en la información de conflicto. Las tres primeras ecuaciones de `obtainBackjumpClause` son claras con respecto a

lo que hemos contado en el párrafo anterior. La primera ecuación representa la condición de parada del algoritmo: llegar a una cláusula con únicamente un literal en el nivel actual. Las dos siguientes representan la aplicación de la regla binaria, teniendo en cuenta que en el literal actual aparece la negación de la correspondiente entrada de la información de conflicto.

Por otro lado, la última ecuación representa el caso de que la última cláusula asociada a la información de conflicto no ha tenido impacto alguno en cómo hemos llegado a contradicción, sino que ha aparecido independientemente mientras se propagaban valores. En este caso, lo único que hacemos es pasar a estudiar la cláusula anterior, sin modificar la cláusula actual.

Veamos la aplicación de este algoritmo siguiendo el ejemplo 3.4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{1 \vee \neg 2 \vee \mathbf{3} \quad 5 \vee 7 \vee \neg \mathbf{3}}{5 \vee 7 \vee 1 \vee \neg \mathbf{2}} \quad \neg 4 \vee 5 \vee \mathbf{2}}{\neg 4 \vee 5 \vee 7 \vee \mathbf{1}} \quad \neg 4 \vee \mathbf{1}}{5 \vee 7 \vee \neg \mathbf{4}} \quad \neg 6 \vee 8 \vee \mathbf{4}}{\neg 6 \vee 8 \vee 7 \vee \mathbf{5}} \quad 8 \vee 7 \vee \neg \mathbf{5}}{8 \vee 7 \vee \neg \mathbf{6}}$$

En el resto de apartados, trabajaremos con el UIP negado en lugar del UIP, pues esta negación corresponde al literal que se deduce de aplicar la regla de `UnitPropagate`. Para obtenerlo, basta buscar aquel literal cuya negación se encuentre en la información de conflicto:

```
op obtainNegatedUIP : ConflictInfoSet Clause → Literal .

eq obtainNegatedUIP(CIS (l → C) Cis, ~(l) \/ D) = ~(l) .
eq obtainNegatedUIP(CIS (~(l) → C) Cis, l \/ D) = l .
```

Nos queda explicar cómo se construye la información de conflicto. Esta información no la hemos incluido a nivel de reglas, para aislar las reglas en sí de otra información adicional que necesitemos manejar. Por tanto, vamos a construir esta información a nivel de estrategias, generando la nueva información según vayamos aplicando las reglas. Esto conducirá a que varias de las comprobaciones de las reglas se tengan que repetir a nivel de estrategias, pues no se puede estudiar cómo se han instanciado los parámetros de la regla desde el nivel de estrategias.

Otro aspecto importante de la información de conflicto es que a nivel de ecuaciones, asumimos que solo contiene la información del último nivel de decisión, mientras que a nivel de estrategias, mantenemos la información de conflicto de todos los niveles del contexto actual.

Esta diferenciación se debe a que los niveles de decisión no son cerrados, en el sentido de que al aplicar la regla de `Backjump`, volvemos a un nivel anterior, el cual se va a extender con nuevos literales propagados. Por tanto, si volvemos a alcanzar un conflicto en este nivel, necesitamos recuperar la información de conflicto de los literales anteriores.

Por ello la manera más adecuada de proceder consiste en almacenar toda la información y restringirnos al nivel actual únicamente cuando vayamos a generar la cláusula de backjump.

Para ello, hemos definido el operador `obtainCurrentConflictInfoSet`, que obtiene la información de conflicto a partir de la última decisión:

```
op obtainCurrentConflictInfoSet : Context ConflictInfoSet → ConflictInfoSet .

ceq obtainCurrentConflictInfoSet(M d(l) N, CIS (l → C) Cis) = (l → C) Cis if
    existsDecisionLiteral(N) = false .
```

Estudiaremos en más detalle la aplicación a nivel de estrategias en la sección 3.3.

3.3. DPLL con aprendizaje a nivel de estrategias

A diferencia del algoritmo DPLL clásico, el algoritmo DPLL con aprendizaje no se puede aplicar a nivel de reglas, pues necesitamos estrategias que nos permitan controlar la obtención de la cláusula de backjump.

Las estrategias del módulo `basic-dpll-strat` se basan en el esquema CDCL, que a su vez es muy parecido al esquema de la figura 2.2. Este esquema se presenta normalmente como un pseudocódigo de un lenguaje de bajo nivel, por lo que vamos a estudiar directamente su adaptación al sistema de reglas. La figura 3.2 recoge el orden de aplicación de reglas.

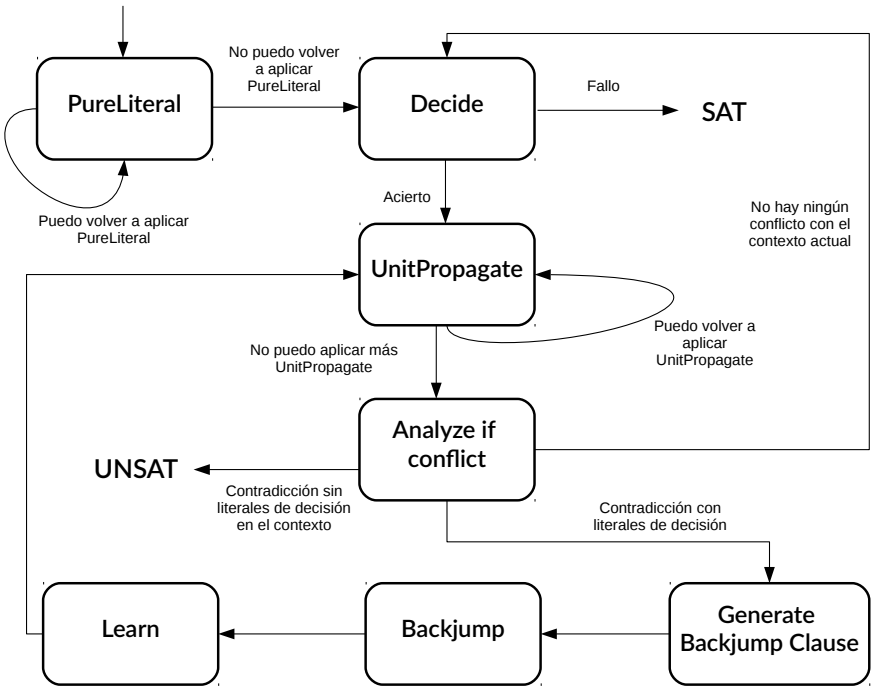


Figura 3.2: Esquema CDCL con aprendizaje de alto nivel

A nivel de Maude, la mayor diferencia con la estrategia anterior es el uso de la estrategia `matchrew` para realizar comprobaciones, así como mantener el parámetro `ConflictInfoSet` coherente con el contexto actual.

Así, la estrategia `success-branch` introduce en la información de conflicto la decisión que hemos tomado:

```
sd success-branch(CIS) := one(Decide) ? matchrew BasicSeq s.t. (N d(l) || CS) := BasicSeq ∧
  CIS' := (CIS l → []) by BasicSeq using (propagate(CIS')) : idle .
```

En la estrategia de `propagate`, ahora necesitamos repetir las comprobaciones de la regla `UnitPropagate` para así hallar cuál es el literal a propagar:

```
sd propagate(CIS) := (matchrew BasicSeq s.t. CTX || (l \/ C), CS := BasicSeq ∧
  CTX |= C == false ∧ definedLiteral(l, CTX) =/= true ∧
  CIS' := CIS l → (l \/ C) by BasicSeq using
  (UnitPropagate[l <- l, C <- C]; propagate(CIS'))) or-else decide-if-fails(CIS) .
```

Nótese que llamamos a `UnitPropagate` directamente con la cláusula de conflicto `C` y el literal `l` que hemos sacado a partir de las comprobaciones, para así aplicar la regla de forma coherente a las comprobaciones previas.

En el momento en el que no podamos aplicar más esta regla, comprobamos si hemos alcanzado o no un conflicto. Esta comprobación se hace ahora a nivel de la estrategia, para así obtener directamente cuál es la cláusula de conflicto en el caso de fallar. En tal caso, generamos directamente la cláusula de backjump y el UIP negado, llamando con ellos a la estrategia `fail-branch`. En caso contrario, volvemos a la estrategia `success-branch`, para así volver a asignar un nuevo literal de decisión.

Esta estrategia nos quedaría de la siguiente forma:

```
sd decide-if-fails(CIS) := matchrew BasicSeq s.t. (M || C, CS) := BasicSeq ∧
  (M |= C == false) ∧ CIS' := obtainCurrentConflictInfoSet(M, CIS) ∧
  C' := obtainBackjumpClause(CIS', C) ∧ l := obtainNegatedUIP(CIS', C') by BasicSeq
  using (fail-branch(l, C, C', CIS)) or-else success-branch(CIS) .
```

Por último, intentamos aplicar cualquiera de las reglas de `Backjump` de forma no determinista, utilizando para ello la cláusula de conflicto, la cláusula de backjump y el UIP negado. En caso de tener éxito, aprendemos la cláusula de backjump y actualizamos la información de conflicto, para que sea coherente

con el salto realizado. Si no podemos aplicar una de estas dos reglas, entonces deducimos que no existe ninguna asignación válida de la fórmula actual.

En Maude, nos quedaría la siguiente estrategia:

```
sd fail-branch(l, C, C', CIS) := (BackjumpWithSeveralDecisions[l <- l, C <- C, C' <- C'] |
  BackjumpWithOneDecision[C <- C, l <- l, C' <- C']) ? (Learn[C <- C'] ; matchrew BasicSeq
  s.t. (M || CS) := BasicSeq ∧ CIS' := generateConflictInfoSetFromBackjump(M, CIS, C')
  by BasicSeq using propagate(CIS')) : Fail .
```

La estrategia inicial que pone en marcha todo el proceso anterior es `basic-dpll-strat`. Esta estrategia realiza el preprocesado de la regla `PureLiteral` e inicializa la información de conflicto del proceso. Su declaración es la siguiente

```
sd basic-dpll-strat := PureLiteral ! ; propagate(emptyCIS) .
```

Finalmente, podemos ejecutar la estrategia siguiendo el mismo razonamiento que hicimos en la sección 2.3:

```
Maude> dsrew [1] in BASIC-DPLL-STRATEGY : emptyCTX || '1 \/ ~('2), ~('1) \/ '2 \/ ~('3), '3
    \/ '2 using basic-dpll-strat .

...

Solution 1
rewrites: 203 in 0ms cpu (0ms real) (~ rewrites/second)
result BasicSequent: d('1) '2 d('3) || '1 \/ ~('2),'2 \/ '3,'2 \/ ~('1),'2 \/ ~('1) \/ ~('3)
```

Capítulo 4

Esquema *Watch-Literal*

Los sistemas de reglas de los capítulos anteriores utilizan la regla `UnitPropagate` para propagar literales. Las estrategias asociadas se basan en aplicar repetidamente esta regla siempre que se pueda, minimizando así el número de decisiones tomadas. Por ello esta regla ocupa aproximadamente el 80 %-90 % del tiempo en los resolutores SAT modernos, buscando cláusulas sobre las que aplicarla. Nos interesa encontrar mecanismos que agilicen este proceso, pues de esta manera tendremos un impacto muy grande en el tiempo total de ejecución.

Uno de los mecanismos más utilizados por los resolutores SAT modernos es el esquema *watch-literal*. Este esquema se introdujo en el resolutor Chaff [8] y se basa en ahorrar comprobaciones innecesarias a la hora de aplicar la regla `UnitPropagate` utilizando cláusulas vigiladas. En la sección 4.1, hablaremos más en detalle sobre cómo funciona.

Este esquema cambia parcialmente el algoritmo DPLL con aprendizaje del capítulo 3. Introduciremos un sistema de reglas no completo propuesto por nosotros mismos, que abstrae este esquema y lo adapta a un sistema de reglas eficiente. Además, aprovecharemos este sistema de reglas para solucionar algunos de los cabos sueltos de los capítulos anteriores, como la comprobación redundante de condiciones de reglas. Lo estudiaremos en detalle en la sección 4.2.

Por último, siguiendo el hilo de la memoria, estudiaremos cuáles son las modificaciones más importantes del nuevo sistema de reglas a nivel de estrategias en la sección 4.3. Las estrategias descritas en este apartado formarán el esqueleto al que amoldaremos las distintas heurísticas basadas en la puntuación y el resolutor Berkmin.

4.1. Explicación del esquema *Watch-literal*

Hemos visto en ejemplos anteriores cómo se aplica la regla `UnitPropagate`. Sin embargo, en todos estos ejemplos se proporcionaba directamente cuál era la cláusula sobre la que se realizaba la propagación, no siendo conscientes del número de operaciones necesarias que hay que llevar a cabo para deducir esta aplicación.

Por otro lado, estos ejemplos trataban cláusulas relativamente cortas, conteniendo la mayoría 2 o 3 literales a lo sumo. Hemos decidido incluir cláusulas de esta longitud para que pudiésemos aplicar rápido la regla `UnitPropagate`. En casos con cláusulas más largas, esta comparación es más costosa, pues necesitamos comparar muchos más literales.

El esquema *watch-literal* funciona increíblemente bien para evitar comparaciones repetidas. La idea es muy sencilla: la regla `UnitPropagate` no se puede aplicar hasta que no tengamos todos los literales negados de la cláusula salvo uno en el contexto actual. Entonces, no es necesario comparar todos los literales de una cláusula: basta comparar el contexto actual con dos literales de esta cláusula que no estén asignados a falso. Si el literal que estamos propagando no es ninguno de ellos, no es necesario llevar a cabo más comprobaciones, pues es claro que nunca podré propagar hasta que asigne la negación de alguno de ellos.

Por ello a partir de ahora hablaremos de cláusulas *vigiladas*, en el sentido de que asociaremos a cada cláusula dos literales que cumplan el invariante anterior. Las cláusulas vigiladas son de la forma $x \vee y : C$, siendo x e y los literales con los que llevar a cabo la comparación y C la cláusula de partida. En Maude, tenemos las siguientes declaraciones:

```
sorts WatchedClause WatchedClauseSet WatchedSequent .
```

```
subsort WatchedClause < WatchedClauseSet .
subsort WatchedSequent < Sequent .

op emptyWCS : → WatchedClauseSet [ctor] .

op _:_ : Clause Clause → WatchedClause [ctor prec 25] .
op _,_ : WatchedClauseSet WatchedClauseSet → WatchedClauseSet
  [ctor assoc id: emptyWCS prec 30] .
op _||_ : Context WatchedClauseSet → WatchedSequent [ctor] .
```

Una sutileza que será muy útil para capítulos posteriores es la declaración del operador `_,_` sin el atributo `comm`. Esto se debe a que queremos recordar el orden en el que fueron generados los literales, de cara al resolutor Berkmin. Esto nos forzará a utilizar más variables a la hora de aplicar encaje de patrones con los conjuntos de cláusulas vigiladas.

Así, en lugar de considerar secuencias básicas como en las secciones anteriores, ahora hablaremos de secuencias vigiladas.

Al partir siempre del contexto vacío, es fácil generar un conjunto de cláusulas vigiladas a partir del conjunto inicial: basta tomar dos literales cualesquiera de estas cláusulas. Asumimos que nuestra fórmula inicial no tiene cláusulas con un solo literal, pues de ser así, podríamos asignar directamente el valor de ese literal al contexto inicial y olvidarnos de esa cláusula.

Supongamos que acabamos de asignar el literal l y queremos actualizar el invariante de todas las cláusulas vigiladas del conjunto. Asumimos que el contexto actual es M y que todas las cláusulas vigiladas cumplían el invariante antes de considerar l. Podemos encontrar los siguientes casos:

1. Si l corresponde a la negación de x o y, entonces quizás necesitemos tomar acciones para mantener el invariante. Por comodidad, asumiremos que $l = \neg x$. De nuevo, nos podemos encontrar otros 3 subcasos:

   a) Se cumple que $y \in M$, por lo que directamente se satisface el invariante. No hace falta llevar a cabo ningún cambio.

   b) Se cumple que $y \notin M$ y que existe otro literal z cuya negación no está en M. Por tanto, puedo reemplazar x por z en la cláusula vigilada, de tal manera que el invariante se siga cumpliendo.

   c) Se cumple que $y \notin M$ y no existe ningún otro literal por el que reemplazar x. Entonces, volveríamos a tener dos casos, según y haya sido previamente asignado o no : $\neg y \notin M$ o $\neg y \in M$. En el primer caso, se sigue que y no está definido en el contexto M, por lo que necesariamente hay que propagar y.

   El segundo caso puede parecer imposible, pues uno supondría que de haber asignado $\neg y$, entonces necesariamente tendría que haber propagado x antes. Sin embargo, esto no siempre es cierto, pues l podría haber propagado previamente a $\neg y$ usando otra cláusula vigilada distinta. En ese caso, la cláusula actual se encuentra *desfasada* en el sentido de que aún no habíamos terminado de propagar l, y por tanto, los invariantes de las cláusulas vigiladas que cambian debido a l no cumplen el invariante.

   En este caso, hemos llegado a una contradicción, por lo que pasaríamos a analizar la causas del conflicto para generar una cláusula de backjump. De hecho, todos los conflictos se detectan al darse este caso, por lo que el esquema *watch-literal* nos proporciona una forma más eficiente de detectar conflictos que el algoritmo DPLL con aprendizaje.

2. En caso contrario, l no dispara la regla `UnitPropagate` ni corresponde a la negación de ninguno de los dos literales, por lo que se mantiene el invariante.

Al proceso de actualizar las cláusulas vigiladas de acuerdo a un literal l lo denominaremos como propagar l.

Esta distinción de casos va a ser vital para plantear la estrategia correspondiente en la sección 4.3. El siguiente ejemplo ilustra todas las posibles situaciones:

Ejemplo 4.1. *Para ver todos los posibles casos del esquema watch-literal, supongamos que tenemos el contexto* $M = 2\ 4\ 1$*, y quiero propagar el literal* $\neg 1$*. Consideramos el conjunto de cláusulas vigiladas*

$$5 \vee 6 : 5 \vee 6 \vee \neg 1\ ,\ 2 \vee \neg 1 : 2 \vee \neg 1 \vee 4\ ,\ 5 \vee \neg 1 : 5 \vee \neg 1 \vee \neg 6\ ,\ \neg 1 \vee 3 : \neg 1 \vee \neg 2 \vee 3\ ,\ \neg 1 \vee \neg 3 : \neg 1 \vee \neg 2 \vee \neg 3$$

En el primer caso, $\neg 1$ *no forma parte del invariante, aunque sí forme parte de la cláusula. No es necesario hacer más comprobaciones ni cambiar el invariante.*

El resto de cláusulas verifican que $\neg 1$ *forma parte de los literales del invariante. Por tanto, nos encontramos en el caso 1 del esquema anterior.*

La cláusula vigilada $2 \vee \neg 1 : 2 \vee \neg 1 \vee 4$ *cumple que* $2 \in M$*, por lo que nos encontramos en el subcaso a) y el invariante ya se cumple.*

Si ahora consideramos la cláusula $5 \vee \neg 1 : 5 \vee \neg 1 \vee \neg 6$*, se verifica que el literal* 6 *no está definido en* M *y podemos actualizar la cláusula vigilada de tal manera que la nueva cláusula vigilada sea de la forma* $5 \vee \neg 6 : 5 \vee \neg 1 \vee \neg 6$*. Se trata del subcaso b).*

En la cláusula vigilada $\neg 1 \vee 3 : \neg 1 \vee \neg 2 \vee 3$ *no puedo reemplazar* $\neg 1$*. Como el literal* 3 *no está definido en* M*, lo podemos añadir al contexto actual aplicando la regla* `UnitPropagate`*. Sin embargo, si ahora intentamos actualizar* $\neg 1 \vee \neg 3 : \neg 1 \vee \neg 2 \vee \neg 3$*, se cumple que* $3 \in M$*, y por tanto, hemos llegado a una contradicción. Estas dos cláusulas representan las posibilidades del subcaso c).*

Cabe preguntarse si el invariante se mantiene cuando aplicamos la regla `Backtrack`. En efecto, cuando aplicamos esta regla, deshacemos una o varias decisiones, por lo que si antes de aplicarla se cumplía que los dos literales no estaban ambos asignados a falso, al tener un contexto más reducido se sigue manteniendo el invariante.

A nivel de implementación en Maude hemos incluido el módulo funcional `WATCH-LITERAL`, que contiene las declaraciones necesarias para llevar a cabo este esquema. Hemos incluido las siguientes operaciones:

```
op changeRepresentative : Context Literal WatchedClause → WatchedClause .

ceq changeRepresentative(M,l, l \/ x : x \/ y \/ l \/ C) = x \/ y : (x \/ y \/ l \/ C)
  if x in M = false ∧ (~(y)) in M = false .

op preprocessWatchedLiterals : ClauseSet → WatchedClauseSet .

eq preprocessWatchedLiterals(emptyCS) = emptyWCS .
eq preprocessWatchedLiterals((x \/ y \/ C), CS) =
  ((x \/ y) : (C \/ x \/ y)) , preprocessWatchedLiterals(CS) .

op obtainValidWatchedLiterals : Clause Context → WatchedClause .

ceq obtainValidWatchedLiterals(x \/ y \/ C, M) = x \/ y : x \/ y \/ C
  if ~(x) in M = false ∧ ~(y) in M = false .
ceq obtainValidWatchedLiterals(x \/ y \/ C, M) = x \/ y : x \/ y \/
  if ~(x) in M = false [owise] .
```

La operación `changeRepresentative` permite sustituir uno de los literales de la cláusula vigilada por otro en el caso de ser posible. La operación `preprocessWatchedLiterals` obtiene dos literales cualesquiera de cada literal de nuestra fórmula de partida para así obtener las cláusulas vigiladas asociadas. Por último, la operación `obtainValidWatchedLiterals` obtiene dos literales válidos de una cláusula cualquiera que mantengan el invariante dado un contexto. Utilizaremos esta operación para obtener la cláusula vigilada asociada a una cláusula de backjump tras ser aprendida. Sabemos que el UIP negado está en el contexto actual tras aplicar la regla `Backjump`, por lo que siempre vamos a poder mantener el invariante.

4.2. Reglas de inferencia con *Watch-Literal*

En los apartados de reglas anteriores, hemos traducido de forma casi literal la definición formal de las reglas de los sistemas de inferencia DPLL a Maude. Esto nos ha conducido a introducir condiciones redundantes que penalizan el tiempo de ejecución, como ya comentamos en la sección 3.3.

En este punto del proyecto, se planteaba un dilema: mantener la completitud de las reglas a costa de perder eficiencia, o perder completitud y forzar a que sean las estrategias las que mantengan la coherencia a la hora de aplicarse. Hemos decidido seguir esta segunda aproximación, para así poder evaluar la componente de eficiencia del motor de Maude frente a un problema clásico.

Además, el sistema de inferencia solo se puede ejecutar utilizando estrategias, por lo que tiene sentido repartir las condiciones entre el nivel de reglas y estrategias según convenga. Discutiremos cómo hemos decidido hacer este reparto al final de la sección 4.3.

Por ello vamos a redefinir todas las reglas de apartados anteriores, incluyendo el esquema *watch-literal* y eliminando condiciones repetidas. Además, vamos a introducir reglas auxiliares que no forman parte

del sistema de inferencia, pero que a nivel de ejecución en Maude nos facilitan el trabajo. Todas estas reglas se encuentran definidas en el módulo `watch-literal-dpll`.

La regla `PreprocessClauseSet` se encuadra dentro de esta idea, y nos permite convertir una secuencia básica en una secuencia vigilada:

```
crl [PreprocessClauseSet] : M || CS ⇒ M || WCS if WCS := preprocessWatchedLiterals(CS) .
```

Vamos a modificar también la regla `PureLiteral`. Esta regla se utiliza para asignar directamente literales al no aparecer su negación en ninguna otra cláusula. Por tanto, sabemos que las cláusulas en las que aparecen los literales asignados de esta forma son satisfechos directamente, por lo que a nivel de resolución del problema, no aportan información alguna. Es por ello que la regla `WPureLiteral` invierte el paradigma: en lugar de asignar literales de esta forma, lo que vamos a hacer es eliminar las cláusulas que contienen a estos literales.

Esto supone que la fórmula a analizar no es exactamente de la que partimos, pero se puede probar que ambas fórmulas son equisatisfactibles: puedo encontrar una asignación válida de una de ellas si y solo si la puedo encontrar de la otra. La idea es sencilla: un asignación del problema con más literales se puede restringir al de menos y viceversa.

En Maude tenemos la siguiente regla:

```
crl [WPureLiteral] : M || (l \/ C), CS ⇒ M || CS if (~(l) in CS) = false ∧
  definedLiteral(l,M) = false .
```

Nótese que la aplicación es a nivel de secuencias básicas y no de secuencias vigiladas. Hemos decidido declararlo así por motivos puramente operacionales, cuya explicación vendrá incluida en el capítulo 5.

Las reglas `WDecide`, `WBackjump`, `WUnitPropagate` y `WFail` están basadas en sus correspondientes declaraciones de la sección 3.1:

```
crl [WDecide] : M || WCS ⇒ M d(l) || WCS if definedLiteral(l,M) = false [nonexec] .
crl [WUnitPropagate] : M || WCS ⇒ M x || WCS if definedLiteral(x,M) = false [nonexec] .
crl [WBackjump1] : M d(x) N d(y) CTX || WCS ⇒ M d(x) N l || WCS
  if existsDecisionLiteral(N) = false ∧ C' := removeLiteralFromClause(l, C) ∧
  M d(x) N |= C' = false ∧ M |= C' =/= false [nonexec] .
crl [WBackjump2] : M d(x) N || WCS ⇒ M l || WCS if existsDecisionLiteral(M) =/= true
  [nonexec] .
crl [WFail] : M || WCS ⇒ failState if existsDecisionLiteral(M) = false .
```

Lo primero que destacan estas reglas es que están declaradas con el atributo `nonexec`. En el caso de `WDecide`, la decisión siempre se tomará a nivel de estrategias, por lo que la única comprobación que se tiene que hacer es que esta decisión no corresponda a un literal ya asignado en el contexto actual.

La regla `WUnitPropagate` también está declarada como `nonexec`, puesto que a nivel de estrategias, haremos la distinción de casos que hemos estudiado en la sección 4.1. Por tanto, en el caso de que necesitemos aplicarla, sabemos que necesariamente se realizará sobre el segundo literal que vigila a la cláusula que estemos considerando. El resto de reglas no tienen diferencias que merezcan la pena reseñar.

Además de las reglas clásicas, también hemos adaptado las reglas del algoritmo DPLL con aprendizaje. Su declaración es la siguiente:

```
crl [WLearn] : M || WCS ⇒ M || WC , WCS if WC := obtainValidWatchedLiterals(C, M) [nonexec]

rl [WForget] : M || WCS, WC, WCS' ⇒ M || WCS, WCS' .

crl [WRestart] : M d(l) N || WCS ⇒ M || WCS if existsDecisionLiteral(M) = false .
```

En el caso de `WLearn`, le pasamos la cláusula de `Backjump` y generamos su correspondiente cláusula vigilada. Esta cláusula siempre se puede generar si la cláusula de backjump tiene al menos dos literales, pues el UIP negado siempre se encuentra en el contexto actual tras realizar el salto.

¿Qué pasa en aquellos casos en el que la cláusula de backjump consiste en un único literal? En estos casos tenemos un problema, pues obviamente no podemos vigilarla con dos literales. Tener una cláusula con un único literal significa que necesariamente ese literal tiene que estar asignado a ese valor. Esto es coherente con la aplicación de la regla `WBackjump2`, pues esta regla añade el literal al nivel 0 de decisión. No tiene sentido añadir esta cláusula al conjunto actual, así que a nivel de estrategias distinguiremos este caso especial.

La regla `WRestart` también difiere de su regla equivalente. En este caso, no reiniciamos el contexto completo, sino que mantenemos todos los literales que se encuentre en el nivel de decisión 0. Esto se

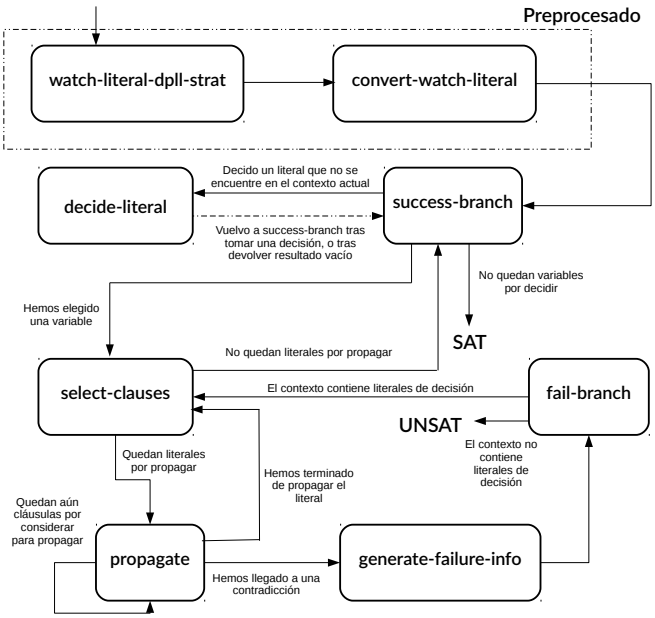


Figura 4.1: Esquema de llamadas entre las estrategias del módulo `watch-literal-dpll-strat`

debe a que estos literales tienen que estar asociados a ese valor obligatoriamente, así que no tiene sentido resetear estos valores.

Por último, hemos añadido una regla extra para poder eliminar varias cláusulas vigiladas de una vez:

```
rl [WChangeWatchedClauseSet] : M || WCS ⇒ M || WCS' [nonexec] .
```

Esta regla será útil para implementar el resolutor Berkmin en el capítulo 6.

4.3. Esquema *Watch-Literal* a nivel de estrategias

Una vez explicados los dos apartados anteriores, el esquema de aplicación de estrategias queda bastante restringido. Este esquema constituye un esqueleto importante para las estrategias de capítulos posteriores, por lo que detallaremos cada una de las estrategias definidas. El flujo de llamadas entre ellas no es fácil de seguir, así que hemos decidido incluir un esquema de llamadas entre distintas estrategias en la figura 4.1. Incluimos explicaciones del flujo en aquellas estrategias que puedan realizar llamadas distintas. Además, se ha añadido a una arista punteada desde la estrategia `decide-literal` a la estrategia `success-branch`. Esta arista se ha añadido para remarcar que la estrategia `decide-literal` no llama a ninguna otra, pero es siempre llamada desde la estrategia `success-branch`, por lo que el flujo retorna a esta última estrategia una vez aplicada.

Antes de explicar cada una de las estrategias del diagrama, queremos remarcar que las dos estrategias correspondientes al preprocesado, `watch-literal-dpll-strat` y `convert-watch-literal`, están definidas sobre secuencias básicas. El resto de estrategias están definidas sobre secuencias vigiladas. Para remarcar esta separación, en el primer caso utilizaremos `BasicSeq` como variable en la declaración de estas estrategias, mientras que en el segundo utilizaremos `WatchedSeq`.

Vamos a detallar una a una cada una de las estrategias declaradas:

- `watch-literal-dpll-strat`: se corresponde a la llamada inicial, en la que eliminamos cláusulas inservibles a través de la estrategia `WPureLiteral !`. En esta llamada, también vamos a almacenar el conjunto de variables que se pueden elegir, que denotaremos por `LiteralSet`. Este conjunto pertenece al sort `PairScoredList`, que explicaremos en el capítulo 5. Hemos decidido utilizar esta representación para así aprovechar las declaraciones en otros módulos de estrategias basados en heurísticas.

```
sd watch-literal-dpll-strat := WPureLiteral ! ; matchrew BasicSeq s.t. (M || CS) :=
    BasicSeq ∧ LiteralSet := obtainInitialScoreList(CS) by BasicSeq using
    convert-watch-literal(LiteralSet) .
```

- `convert-watch-literal`: en esta estrategia obtenemos el conjunto de cláusulas vigiladas con el que vamos a trabajar a partir de la fórmula de partida.

  ```
  sd convert-watch-literal(LiteralSet) := PreprocessClauseSet ;
     success-branch(emptyCIS, LiteralSet) .
  ```

- `success-branch`: esta estrategia representa la fase de decisión. Primero llamamos a la estrategia `decide-literal` para obtener el siguiente literal de decisión, de tal forma que si esta estrategia falla, entonces hemos llegado a la situación en la que no quedan variables de decisión y la asignación actual es válida. En caso de obtener un literal de decisión, actualizamos la información de conflicto y pasamos a seleccionar las cláusulas. Para realizar la propagación de esta decisión, vamos a tomar el literal negado que acabamos de decidir. Esto se debe a que, según lo estudiado en la sección 4.1, las únicas cláusulas vigiladas que son candidatas a ser modificadas son aquellas que contienen al literal negado vigilando la cláusula. Es más cómodo hacer las comprobaciones directamente con este literal.

  ```
  sd success-branch(CIS, LiteralSet) := decide-literal(LiteralSet) ?
     matchrew WatchedSeq s.t. (M d(l) || WCS) := WatchedSeq ∧ CIS' := (CIS l → [])
     ∧ x := ~(l) by WatchedSeq using (select-clauses(x, CIS', LiteralSet)) : idle .
  ```

- `decide-literal`: la estrategia `decide-literal` es una estrategia recursiva que intenta asignar el primer literal de decisión de la lista de literales. La estrategia puede no devolver ningún resultado, pues no hay una ecuación que encaje con la constante `emptySPL`, que es la lista vacía de literales.

  Esta estrategia no llama a ninguna otra, pues estamos interesados únicamente en asignar un literal válido en caso de ser posible, o no devolver ningún resultado en caso contrario.

  ```
  sd decide-literal(l :| F , LiteralSet) := WDecide[l <- l] or-else
     decide-literal(LiteralSet) .
  ```

- `select-clauses`: la estrategia `select-clauses` se utiliza para propagar un literal. Para ello, antes de actualizar la información de las cláusulas vigiladas, vamos a obtener todas aquellas que son candidatas a sufrir modificaciones. Este paso no es necesario, pero así evitamos repetir la búsqueda de las cláusulas vigiladas cada vez que volvamos a intentar propagar el mismo literal.

  Esta estrategia tiene en cuenta que es posible que debido a propagaciones previas, nos hayan surgido dos o más propagaciones que realizar. Debido a ello, esta estrategia almacena una lista de cláusulas por propagar, de tal manera que cuando se termine de propagar un literal de la lista, pasamos a analizar el siguiente.

  ```
  sd select-clauses(emptyCTX, CIS, LiteralSet) := success-branch(CIS, LiteralSet) .

  sd select-clauses(l CTX, CIS, LiteralSet) := (matchrew WatchedSeq s.t.
     M || WCS := WatchedSeq ∧ CS := obtainCandidateClauses(l, WCS)
     by WatchedSeq using propagate(l CTX, CIS, CS, LiteralSet)) .
  ```

- `propagate`: dado el conjunto de cláusulas vigiladas que contienen a l como uno de los dos literales, la estrategia `propagate` analiza el posible caso del esquema *watch-literal*:

  1. Si la condición de `matchrew` falla, necesariamente se debe a que `l in M` o `x in M` son ciertas. Por tanto, nos encontraríamos en el caso a) y pasaríamos directamente a estudiar la siguiente cláusula vigilada.
  2. Si la estrategia `WUpdateClause[C <- C, WC <- WC]` tiene éxito, entonces hemos podido reemplazar el literal `l`. Nos encontraríamos por tanto en el caso b), y pasaríamos a estudiar la siguiente cláusula.
  3. Si la estrategia anterior no ha tenido éxito, probamos a aplicar la regla `WUnitPropagate` sobre el literal x, tal y como se corresponde a la primera parte del subcaso c). En caso de tener éxito, seguimos estudiando la siguiente cláusula y añadimos $\neg x$ al conjunto de cláusulas a propagar.

4. Por último, si la estrategia anterior también falla, entonces necesariamente hemos llegado a una contradicción. Sabemos que la cláusula conflictiva es C, así que llamamos a `generate-failure-info` directamente con esta información.

```
sd propagate(l CTX, CIS, emptyCS, LiteralSet) := select-clauses(CTX, CIS, LiteralSet) .

sd propagate(l CTX, CIS, (C, CS), LiteralSet) := (matchrew WatchedSeq s.t.
    M || WCS, (l \/ x) : C , WCS' := WatchedSeq ∧ l in M =/= true ∧ x in M =/= true ∧
    WC' := (l \/ x) : C ∧ WC := changeRepresentative(M, l, WC') ∧
    CIS' := (CIS x → C) ∧ y := ~(x) by WatchedSeq
    using ((WUpdateClause[C <- C, WC <- WC] ;
    propagate(l CTX, CIS, CS, LiteralSet)) or-else (WUnitPropagate[x <- x] ;
    propagate(l CTX y, CIS', CS, LiteralSet)) or-else
    generate-failure-info(CIS, C, LiteralSet) )) or-else
    propagate(l CTX, CIS, CS, LiteralSet) .
```

- `generate-failure-info`: la estrategia `generate-failure-info` obtiene la información de conflicto del nivel actual y genera la cláusula de backjump a partir de ella. Le pasa esta información a la estrategia `fail-branch`.

```
sd generate-failure-info(CIS, C, LiteralSet) := matchrew WatchedSeq
    s.t. (M || WCS) := WatchedSeq ∧ CIS' := obtainCurrentConflictInfoSet(M, CIS)
    ∧ C' := obtainBackjumpClause(CIS', C) ∧ l := obtainNegatedUIP(CIS', C')
    by WatchedSeq using (fail-branch(l, C', CIS, LiteralSet)) .
```

- `fail-branch`: la estrategia `fail-branch` distingue si la cláusula de backjump generada en la fase anterior tiene longitud uno o no. En caso afirmativo, aplicamos directamente la regla `WBackjump2` y volvemos a intentar propagar el nuevo literal añadido al contexto. Si la regla `WBackjump2` no tiene éxito, es que necesariamente no hay ningún literal de decisión en el contexto actual y deducimos que no existe ninguna asignación válida.

  El caso contrario es muy similar, salvo que intentamos aplicar primero `WBackjump` y luego `WBackjump2`; y en caso de tener éxito, añadimos la cláusula de backjump al conjunto de cláusulas vigiladas.

```
sd fail-branch(l, l, CIS, LiteralSet) := WBackjump2[l <- l] ? matchrew WatchedSeq
    s.t. (M x || WCS) := WatchedSeq ∧ CTX := M x ∧ y := ~(x)
    ∧ CIS' := generateConflictInfoSetFromBackjump(CTX, CIS, l)
    by WatchedSeq using select-clauses(y, CIS', LiteralSet) : WFail .

sd fail-branch(l, C, CIS, LiteralSet) := (WBackjump1[l <- l, C <- C] or-else
    WBackjump2[l <- l]) ? (WLearn[C <- C] ;
    matchrew WatchedSeq s.t. (M x || WCS) := WatchedSeq ∧ CTX := M x ∧ y := ~(x)
    ∧ CIS' := generateConflictInfoSetFromBackjump(CTX, CIS, C)
    by WatchedSeq using select-clauses(y, CIS', LiteralSet)) : WFail .
```

Una vez hemos analizado todas las estrategias, vamos a explicar el criterio que se ha seguido a la hora de definir las condiciones de las reglas.

La idea fundamental es recoger en el nivel de estrategias aquellas condiciones de reglas que contengan un elemento del cual necesitemos extraer información, pero que no podemos deducir después de haber aplicado esta regla.

Esto se hace explícito en el caso de la estrategia `propagate`, en la que necesitamos saber cuál es la cláusula vigilada que vamos a actualizar, para así extraer el otro literal que vigila a esta cláusula y hacer la distinción de casos. Si aplicase la regla `WUpdateClause` sin hacer un `matchrew` previo en el que instanciase esta información, entonces en caso de fallo de esta regla no sabría en cuál de los tres casos restantes me encontraría, al no saber siquiera cuál es la cláusula de conflicto con la que ha fallado.

Dado este caso concreto, nos podríamos preguntar por qué no declarar entonces todas las reglas incondicionales y hacer todas las comprobaciones a nivel de estrategias. Este sería un planteamiento válido, pero en ciertas situaciones nos volvería a llevar a tener que plantear estrategias más complejas que utilicen más comprobaciones.

Volviendo al caso de la estrategia `propagate` y la regla `WUpdateClause`, imaginemos que la condición de matching de `WUpdateClause` se hace en la declaración de `matchrew` previa, junto con el resto de condiciones.

En este caso, si la estrategia de `matchrew` falla, entonces no sabemos si se debe a la condición `x in M = false` o a la condición de que la variable `WC` está en el *kind* de su tipo. En el primer caso, nos encontraríamos en el caso a) del esquema *watch-literal*, mientras que el segundo, correspondería al caso b). No habría entonces forma de distinguir estos dos casos utilizando un único `matchrew`.

Resumiendo, si queremos evitar comparaciones repetidas entre reglas y estrategias, entonces tenemos que reflexionar sobre el nivel en el que incluir las comprobaciones del sistema de reglas. Un razonamiento que suele funcionar bien es el referido anteriormente: llevar al nivel de estrategias aquellas condiciones que involucran términos que no se pueden obtener desde el nivel de estrategias, y son fundamentales para su correcto funcionamiento.

En el caso de tener un sistema de inferencia completo sin necesidad de estrategias, es recomendable mantener las condiciones en el nivel de reglas, aunque sean redundantes. De esta manera, podremos seguir reescribiendo términos sin necesidad de recurrir al nivel de estrategias.

Sin embargo, si el sistema necesita de estrategias para poder aplicar reglas, no es descabellado repartir condiciones entre ambos niveles, dado que necesariamente ambos están obligados a coexistir. En el caso de los módulos `watch-literal` y `watch-literal-dpll`, hemos visto cómo un reparto razonado de las mismas nos evita tener que repetir comprobaciones.

Esta situación se puede extender a otros ámbitos de Maude. La separación entre niveles no es siempre tan clara y muchas veces nos vemos obligados a tomar decisiones con respecto a qué nivel incluir ciertas declaraciones que a la larga pueden dificultar las implementaciones. No existe un razonamiento globalmente válido para abordar este problema: generalmente, es la experiencia del usuario con Maude la que le provee de este conocimiento.

Capítulo 5

Heurísticas basadas en puntuación

En los capítulos anteriores, nos hemos centrado en la parte formal del algoritmo DPLL, estudiando distintos sistemas de reglas y estrategias. Sin embargo, no hemos entrado apenas en detalle en la parte de heurísticas, que es otro de los aspectos fundamentales de los resolutores SAT modernos.

Uno de los usos más comunes de heurística en el algoritmo DPLL corresponde a la elección de un literal para aplicar la regla `Decide`. Tomar literales prometedores suele conducir a evitar hacer demasiadas exploraciones en el árbol de derivación, lo que tiene un impacto muy grande en la eficiencia de las implementaciones actuales.

Muchas de estas heurísticas se basan en asignar puntuaciones a cada literal, de tal manera que aquellos literales con mayor puntuación se corresponden con aquellos literales que debemos asignar antes. Estas heurísticas forman parte de las heurísticas basadas en puntuación, que pueden ser de dos tipos: estáticas, si no se modifica la puntuación de los literales una vez obtenida; y dinámicas, en caso contrario. Por lo general, las heurísticas dinámicas obtienen mejores resultados que las estáticas, al ser capaz de adaptarse a la información que vamos generando en el proceso. Sin embargo, en algunos casos su cómputo puede ser demasiado elevado y no compensar el tiempo que ahorramos en explorar el árbol.

Nosotros vamos a centrarnos en introducir dos heurísticas: *Jeroslow-Wang* (JW) y *Variable State Independent Decaying Sum* (VSIDS). Hemos elegido estas heurísticas para así analizar un caso de heurística estática y otro de heurística dinámica. Además, estas heurísticas destacan dentro de sus categorías por ser eficientes y simples de implementar.

Antes de empezar a analizar ambas heurísticas, vamos a introducir el módulo funcional `SCORED-HEURISTIC`. Este módulo contiene las definiciones comunes en las que basaremos nuestras implementaciones de heurísticas basadas en puntuación. Hemos incluido los siguientes tipos y constructores:

```
sorts ScorePair ScorePairList .
subsort ScorePair < ScorePairList .

op emptySPL : → ScorePairList [ctor] .
op _:|_ : Literal Float → ScorePair [ctor prec 20].
op _,_ : ScorePairList ScorePairList → ScorePairList
  [ctor assoc id: emptySPL prec 30] .
```

Un invariante que vamos a pedir al tipo `ScorePairList` es que mantenga los literales ordenados por orden descendiente de puntuación, para así facilitar el proceso de búsqueda del literal con valor más alto no asignado. En este punto, podríamos haber utilizado el módulo predefinido de Maude que permite trabajar con listas genéricas que se pueden ordenar. Sin embargo, esto nos llevaría a tener que ejecutar la ecuación `sort` cada vez que quisiésemos actualizar el orden, volviéndose el proceso más costoso. Por ello hemos decidido crear nuestro propio tipo de datos con operaciones más eficientes que asumen que la lista de partida ya está ordenada. La ecuación principal de este módulo es `insertScorePair`, que inserta un par no incluido en la lista, manteniendo el orden:

```
op insertScorePair : ScorePair ScorePairList → ScorePairList .

eq insertScorePair(l :| N, emptySPL) = l :| N .
ceq insertScorePair(l :| N, (l1 :| N1 , SPL1)) = l :| N , l1 :| N1 , SPL1 if N1 < N .
ceq insertScorePair(l :| N, (SPL1 , l1 :| N1) ) = SPL1 , l1 :| N1 , l :| N if N ≤ N1 .
ceq insertScorePair(l :| N, (SPL1, l1 :| N1 , l2 :| N2, SPL2)) =
  SPL1 , l1 :| N1, l :| N, l2 :| N2, SPL2 if N ≤ N1 ∧ N2 < N .
```

Estas ecuaciones se basan en buscar la posición del nuevo par dentro de la lista a partir de comparaciones con los elementos de la misma. La elección de los operadores $<$ y $\leq$ en estas ecuaciones se ha hecho para garantizar la confluencia de las ecuaciones, de manera que se introduce el par en la posición más a la derecha de todas las posibles en caso de empate.

Si ahora queremos modificar alguna puntuación, basta con sacar el elemento que queremos modificar de la lista utilizando encaje de patrones e introducirlo de nuevo con la nueva puntuación que le hemos asociado:

```
op modifyScore : Literal Float ScorePairList → ScorePairList .

eq modifyScore(l, N, (SPL1, l :| M, SPL2)) = insertScorePair(l :| N, (SPL1, SPL2)) .
```

Hemos definido una serie de operadores auxiliares que nos van a ser muy útiles para implementar los métodos de los apartados posteriores. Entre ellos, destacamos los siguientes:

```
op addScore : Literal Float ScorePairList → ScorePairList .

eq addScore(l, N, (SPL1, l :| M, SPL2)) = insertScorePair(l :| (N + M), (SPL1, SPL2)) .

op divideScorePairList : ScorePairList Float → ScorePairList .

eq divideScorePairList(emptySPL, N) = emptySPL .
eq divideScorePairList((l :| M , SPL1), N) = (l :| (M / N)) , divideScorePairList(SPL1, N) .

op updateClauseScore : Clause ScorePairList Float → ScorePairList .

eq updateClauseScore([], SPL1, N) = SPL1 .
eq updateClauseScore(l \/ C, SPL1, N) = updateClauseScore(C, addScore(l, N, SPL1), N) .

op updateClauseSetScore : ClauseSet ScorePairList Float → ScorePairList .

eq updateClauseSetScore(emptyCS, SPL1, N) = SPL1 .
eq updateClauseSetScore((C, CS), SPL1, N) =
  updateClauseSetScore(CS, updateClauseScore(C, SPL1, N), N) .
```

En el caso del operador `updateClauseScore`, sumamos una cantidad `N` a cada literal asociado a la cláusula `C`.

5.1. Jeroslow-Wang

La heurística *Jeroslow-Wang* [6] o JW es una de las heurísticas estáticas más utilizadas y reconocidas dentro de su categoría. Se basa en la idea intuitiva de que los literales más prometedores son aquellos que aparecen en más cláusulas, pues de esta forma, conseguiré que se satisfagan más cláusulas con una decisión. La heurística JW lleva este razonamiento un paso más allá, teniendo en cuenta también el tamaño de las cláusulas en las que interviene cada literal. Así, se entiende que las cláusulas que son más cortas aportan más información de cara a ser elegidas, pues facilitan la aplicación de la regla `UnitPropagate` y por tanto se toman menos decisiones.

La puntuación de las variables viene dada por la siguiente fórmula:

$$p(l) = \sum_{l \in C, C \in F} 2^{-|C|}$$

donde $|C|$ representa el tamaño de la cláusula.

Esta heurística tendría también su contraparte dinámica, en el caso de actualizar la puntuación cada vez que aprendemos una nueva cláusula. En nuestro caso, hemos trabajado con la versión estática.

En Maude, la implementación de esta heurística se incluye en el módulo `JW-HEURISTIC`. Su implementación es muy sencilla utilizando los operadores introducidos anteriormente:

```
op JWHeuristicClauseSet : ClauseSet ScorePairList → ScorePairList .

eq JWHeuristicClauseSet(emptyCS, SPL) = SPL .
eq JWHeuristicClauseSet((C,CS) , SPL) = JWHeuristicClauseSet(CS, updateClauseScore(C, SPL, 2
    .0 ^ (-(float(csize(C)) )))) .
```

```
op JWHeuristic : ClauseSet → ScorePairList .

eq JWHeuristic(CS) = JWHeuristicClauseSet(CS, obtainInitialScoreList(CS)) .
```

A nivel de estrategias, vamos a reaprovechar las estrategias del módulo `WATCH-LITERAL-DPLL-STRATEGY` explicadas en la sección 4.3. En esta sección, ya utilizamos el tipo `ScorePairList` para llevar a cabo las decisiones, por lo que únicamente tenemos que definir una nueva estrategia inicial que precompute esta heurística:

```
sd jw-heuristic-strat := WPureLiteral ! ; matchrew WatchedSeq s.t. (M || CS) := WatchedSeq ∧
  LiteralSet := JWHeuristic(CS) by WatchedSeq using convert-watch-literal(LiteralSet) .
```

Cabe destacar que eliminamos las cláusulas que no aportan información con `WPureLiteral` antes de computar la heurística JW.

5.2. Variable State Independent Decaying Sum

La heurística VSIDS es una de las heurísticas clásicas que se introducen para el estudio del problema SAT en los resolutores modernos. Esta heurística fue incluida en el resolutor Chaff [8] y fue una de las primeras heurísticas que se basó en la edad de las cláusulas para ponderar las puntuaciones.

Su funcionamiento es el siguiente: al principio, inicializamos el contador de todos los literales al número de apariciones en las cláusulas iniciales. Cada vez que añadimos una cláusula nueva, se incrementa el contador de cada literal asociado a esta cláusula. La clave de esta heurística es que periódicamente dividimos todas las puntuaciones por una misma constante. Así conseguimos favorecer a los literales que aparecen en las cláusulas añadidas más recientemente a la hora de tomar una decisión.

El problema con dividir periódicamente es que todas las cláusulas que hemos aprendido tras la última actualización tienen el mismo peso y por tanto las cláusulas aprendidas más recientemente no destacan sobre estas. Tampoco podemos dividir muy frecuentemente, pues en este caso todos los contadores acabarían conteniendo valores cercanos a 0 y solo nos aportarían información sobre las últimas cláusulas aprendidas.

En Maude la declaración de la heurística se recoge en el módulo `VSIDS-HEURISTIC`, que contiene las siguientes operaciones:

```
op decaySum : ScorePairList → ScorePairList .

eq decaySum(SPL) = divideScorePairList(SPL, 2.0) .

op initializeVSIDS : ClauseSet → ScorePairList .

eq initializeVSIDS(CS) = updateClauseSetScore(CS, obtainInitialScoreList(CS), 1.0) .
```

Hemos decidido dividir por 2 cada 100 decisiones tomadas. Estos parámetros se han fijado de acuerdo al tamaño de las cláusulas con las que estamos trabajando, aunque no podemos garantizar que esta elección sea óptima.

A diferencia de la heurística JW, en este caso no podemos reutilizar las declaraciones del módulo `WATCH-LITERAL-DPLL-STRATEGY`, pues necesitamos incluir un parámetro adicional en la declaración de las estrategias que almacene el número de decisiones tomadas hasta el momento.

Hemos definido las estrategias para aplicar esta heurística en el módulo `VSIDS-HEURISTIC-STRATEGY`. Las estrategias declaradas son esencialmente las mismas que discutimos en la sección 4.3, salvo que están declaradas con un parámetro extra. Además, hemos añadido una nueva estrategia que actualiza el valor de los contadores cuando hemos alcanzado las 100 decisiones: `decide-if-sum-decay`. Su declaración en Maude es la siguiente:

```
sd decide-if-sum-decay(CIS, CurrentLiterals, ScoredLiterals, 100) := matchrew WatchedSeq
  s.t. M || WCS := WatchedSeq ∧ NewScoredLiterals := decaySum(ScoredLiterals) by WatchedSeq
  using success-branch(CIS, NewScoredLiterals, NewScoredLiterals, 0) .

csd decide-if-sum-decay(CIS, CurrentLiterals, ScoredLiterals, N) :=
  success-branch(CIS, CurrentLiterals, ScoredLiterals, N + 1) if N < 100 .
```

Esta estrategia es llamada después de hacer una decisión, desde la estrategia `decide-literal`. El resto de declaraciones se mantiene similar.

Capítulo 6

Resolutor SAT completo: Berkmin

A lo largo de esta memoria, hemos presentado distintos sistemas de inferencia, estrategias, técnicas y heurísticas para mejorar el rendimiento de los resolutores SAT. Hemos elegido desglosar claramente en distintos capítulos cada uno de estos aspectos, para así hacer más entendible al lector las ideas que subyacen a los mismos. No obstante, los resolutores SAT actuales combinan distintas heurísticas y técnicas para alcanzar un mayor rendimiento.

Nuestro propósito en este capítulo es combinar todo el conocimiento acumulado hasta este punto para así generar finalmente una implementación completa de un resolutor SAT. Además, se incluye el único punto que habíamos dejado pendiente en la memoria: heurísticas efectivas para aplicar las reglas `Forget` y `Restart`.

Hemos decidido implementar el resolutor Berkmin, al estar perfectamente detalladas las distintas ideas y estructuras que utiliza en el artículo [5].

Además, esta implementación pretende servir de base para futuros desarrollos de resolutores SAT en Maude. A nivel de estrategias y reglas, los cambios van a ser mínimos con respecto a versiones anteriores, así que nos centraremos en los requerimientos del módulo funcional `berkmin-heuristic`.

Para ello, en las secciones 6.1 y 6.2 explicaremos brevemente los distintos aspectos de este resolutor. Por último, discutiremos algunos detalles de implementación en la sección 6.3, haciendo especial hincapié en aquellos aspectos que creemos que son relevantes a tener en cuenta para adaptar otros resolutores. A estas alturas del trabajo, asumimos que el lector está familiarizado con la filosofía de definición de ecuaciones, reglas y estrategias en Maude; así que vamos a obviar los pormenores de la implementación.

6.1. Toma de decisiones

Las heurísticas que emplea Berkmin a la hora de tomar decisiones son bastante más complicadas que las introducidas en el capítulo 5. A grandes rasgos, Berkmin combina 3 heurísticas basadas en puntuación de literales para decidir cuál es el siguiente literal a elegir.

Antes de explicar cómo funcionan estas heurísticas, necesitamos entender primero de qué manera se realiza la gestión de cláusulas en Berkmin. La implementación de Berkmin considera a este conjunto como una pila, en el que las cláusulas más recientes se encuentran en la parte superior de la misma. Además, se almacena un puntero que apunta a la cláusula más reciente no satisfecha.

Esta cláusula se utiliza para elegir el siguiente literal a decidir: solo podremos elegir uno de los literales que se encuentran en esta cláusula o la negación de las mismas. De esta manera, Berkmin se centra en las cláusulas más recientes, pues las últimas cláusulas analizadas están más fuertemente relacionadas con la rama del árbol que estamos explorando y nos permite deducir información más centrada en ciertos literales.

El mecanismo de decisión de Berkmin se divide en 2 fases: primero elegimos una variable que tiene un literal asociado que aparece en esta cláusula, y después elegimos cuál de los dos literales asociados a esa variable se toma como decisión. De esta manera, si elegimos la negación del literal que aparece en esta cláusula, mantendremos esta cláusula sin satisfacer. Sin embargo, sabemos que terminará siendo satisfecha en algún punto ya que en caso de elegir todos los literales negados salvo uno, necesariamente este último literal se asignará a cierto tras la aplicación de la regla `UnitPropagate`.

Para decidir qué variable elegir, Berkmin lleva un contador de actividad de variable *var_activity*. Este contador se inicializa a 0 para cada variable en nuestra fórmula, y se actualiza según generamos cláusulas de backjump. En lugar de utilizar únicamente esta cláusula para actualizar las puntuaciones, Berkmin

considera cada cláusula intermedia que ha intervenido en la generación de la cláusula de backjump, salvo la propia cláusula de conflicto. Se suma uno a una variable por cada literal asociado que aparece en estas cláusulas.

Así, por ejemplo, en el caso del ejemplo 3.2, las cláusulas a considerar son

$$\neg 1 \vee \neg 8 \vee 9 \ , \ \neg 7 \vee 8 \ , \ \neg 1 \vee \neg 4 \vee \neg 7$$

por lo que aumentamos en 1 la actividad de las variables 4 y 9, y en 2 las variables $1, 7$ y 8.

La idea de esta heurística es favorecer aquellas variables que han estado involucradas en la generación de varios conflictos, pues así podemos seguir generando más información basada en ellas. Este planteamiento favorece la aparición de conflictos cada vez con mayor información de cara a dejar de explorar ciertas ramas.

Una vez hemos elegido una variable, tenemos que elegir cuál de sus literales asociados nos interesa. Esta selección se hace de manera independiente a la decisión de la variable porque la elección de un literal o su negado afecta sobre cuál de las dos ramas del árbol queremos explorar primero, mientras que con la elección de la variable elegimos qué niveles del árbol queremos generar primero.

Dependiendo de si la cláusula considerada actualmente es una cláusula de backjump o pertenecía al conjunto inicial, tenemos dos posibles heurísticas.

En el caso de ser una cláusula de backjump, utilizamos el contador *lit_activity*. Este contador es muy similar al de *var_activity*, salvo que considera únicamente la cláusula de backjump a la hora de actualizar la puntuación. Esta función pretende eliminar la asimetría que se puede generar a la hora de explorar ramas del árbol de derivación debido a los reinicios selectivos. Si al reiniciar elegimos recurrentemente los mismos literales, entonces exploraremos las mismas ramas y no podremos explorar nuevas.

De esta manera, si aplicamos la regla `Decide` con l, entonces las cláusulas de conflicto que podamos encontrarnos a partir de esta decisión necesariamente contendrán a $\neg l$. La puntuación de *lit_activity*(l) se mantendrá mientras que la de *lit_activity*$(\neg l)$ puede aumentar, pudiéndose alternar así entre las dos decisiones. Este equilibrio es muy importante sobre todo a la hora de probar que un problema es insatisfactible, pues generalmente necesitamos explorar ambas posibilidades asociadas a l y probar que ninguna de sus ramas asociadas es satisfactible.

Si la cláusula más reciente sin satisfacer corresponde al conjunto de cláusulas iniciales, entonces se utiliza otro contador distinto llamado *nb_two*. Este contador se corresponde con una heurística estática y mide el número de cláusulas binarias con las que está relacionada un literal. Este contador se computa de la siguiente forma: contamos el número de cláusulas binarias de la forma $l \vee x$, y por cada una de estas cláusulas, sumamos el número de cláusulas binarias en las que aparece $\neg x$. Con este contador se pretende medir el número de veces que podemos aplicar la regla `UnitPropagate` si asignamos el literal l a falso.

En nuestra implementación de Maude, hemos decidido no computar esta función. La razón de ello es que nuestros experimentos abordan instancias del problema 3-SAT y por tanto el valor de este contador siempre será nulo al no existir cláusulas binarias. En lugar de ello, hemos sustituido esta heurística estática por la otra heurística estática que hemos trabajado previamente: *JW*. Los detalles de esta heurística se encuentran en la sección 5.1.

6.2. Eliminación de cláusulas y reinicios selectivos

Los procesos de eliminación de cláusulas y reinicio se realizan de forma paralela en Berkmin, así que vamos a estudiar ambos de forma conjunta.

Berkmin vuelve a hacer uso de la edad de las cláusulas a la hora de eliminarlas. Para ello, utiliza un contador llamado *clause_activity(C)* que mide el número de conflictos en los que ha estado involucrada la cláusula C. Si una cláusula ha estado involucrada en muchos conflictos, asumimos que esa cláusula es interesante de cara a generar futuros conflictos. Por esta razón, nos interesa mantener esta cláusula en la fórmula.

A la hora de realizar el borrado de cláusulas aprendidas, Berkmin considera dos subconjuntos de cláusulas: aquellas cuya distancia al elemento superior de la pila es menor que $\frac{1}{16}$ veces el número de aprendidas, y aquellas que se encuentran a distancia mayor. Al primer subconjunto se le considera el conjunto de cláusulas nuevas, y al segundo, el conjunto de cláusulas antiguas.

Mantendremos un elemento del conjunto de cláusulas nuevas si su tamaño es menor que 43 o *clause_activity(C)* > 7. En el caso de las cláusulas antiguas, mantendremos una cláusula si su tamaño es menor que 7 o *clause_activity(C)* $>$ *activity_threshold*, siendo este último una variable que inicializamos

a 10 y aumentamos cada vez que eliminamos cláusulas. De esta manera, si alguna cláusula fue muy utilizada al inicio del proceso pero luego no vuelve a intervenir, se consigue eliminar en algún punto.

Este proceso se repite periódicamente tras generar 150 cláusulas de backjump. En apartados anteriores, afirmamos que la aplicación de la regla `Restart` se tenía que realizar de forma monótona, principio que estamos violando con esta implementación. Los autores de Berkmin afirman que empíricamente han comprobado que solo necesitan mantener la última cláusula aprendida para no entrar en bucles y garantizar la obtención de resultados válidos. Nosotros hemos decidido seguir este planteamiento, aunque no sería complicado modificarlo para aumentar este límite cada vez que reiniciemos.

Las constantes que aparecen en este apartado se han tomado basándose en las referencias de Berkmin y los propios experimentos con la implementación de Maude, que se detallan en el capítulo 7.

6.3. Implementación en Maude

Este apartado pretende plantear las cuestiones que uno debe plantearse a la hora de construir una implementación de un resolutor SAT completo.

El primer paso recomendable a la hora de abordar esta cuestión consiste en identificar toda la información necesaria para ser capaces de aplicar todas las heurísticas y pensar en una representación válida de cada una de las estructuras que intervienen con la que podamos trabajar a nivel de Maude.

En el caso del resolutor Berkmin, es claro que al menos necesitamos almacenar los contadores *var_activity*, *lit_activity*, *nb_two* y *clause_activity*. Además, sabemos que a nivel de reglas trabajaremos con secuencias que contienen el contexto y la fórmula actual, por lo que no es necesario repetir esta información. Para simular el puntero que apunta a la cláusula más reciente no satisfecha, vamos a guardar directamente esta cláusula.

Para distinguir cuál de los dos contadores utilizar, si *lit_activity* o *nb_two*, necesitamos determinar si la cláusula actual ha sido aprendida o no. Para ello, vamos a simular una numeración para las cláusulas, identificando con 0 la más antigua. Si la cláusula que estamos considerando tiene un número asociado mayor que el número de cláusulas iniciales, entonces claramente esta cláusula es aprendida. Por tanto, vamos a llevar dos contadores: uno que almacene el número de cláusulas iniciales y otro con la posición de la cláusula actual.

También necesitamos saber el número de cláusulas aprendidas para saber hasta qué punto podemos borrar cláusulas de la fórmula. Esto lo simularemos con otro contador que registre el número de cláusulas aprendidas hasta el momento. También almacenaremos una variable con el umbral de actividad para borrar cláusulas antiguas.

Por último, necesitaremos guardar el número de cláusulas de backjump generadas para decidir si aplicar el borrado y reseteo o no.

El segundo paso consiste en identificar cómo realizar el reparto de información entre los distintos niveles.

El nivel de reglas debe considerar únicamente secuencias del tipo $M \mid\mid F$. Esto nos permite aislar completamente la información esencial que se utiliza para declarar el sistema de inferencias del resto de estructuras que guían la aplicación de reglas. El mayor beneficio que conseguimos de esta aproximación es la posibilidad de reutilizar el mismo sistema de inferencia para distintos resolutores sin tener que hacer cambios. De hecho, las estrategias basadas en las heurísticas JW, VSIDS y Berkmin se han construido sobre el módulo de reglas `WATCH-LITERAL-DPLL`, pese a basarse en planteamientos muy distintos.

El reparto del resto de estructuras auxiliares lo haremos entre el nivel de ecuaciones y reglas. Por lo general, se deben englobar todas estas estructuras en un solo tipo a nivel de ecuaciones, que iremos actualizando a nivel de estrategias. La única información que podríamos dejar fuera del nivel de ecuaciones es aquella que directamente se utilice para la declaración de estrategias y sea fácil de manejar.

En nuestro caso, hemos declarado el tipo `BerkminInfo`, que incluye todas las estructuras anteriores, salvo el contador de cláusulas de backjump generadas. Veamos su declaración e inicialización:

```
op <_,_,_,_,_,_,_,_,_,_> : ScorePairList ScorePairList ScorePairList Nat Nat Clause
  Nat ClauseScoreList Nat → BerkminInfo [ctor] .

op initializeBerkmin : ClauseSet → BerkminInfo .

ceq initializeBerkmin(C,CS) = < initializeVarActivity(CS'),
  initializeLitActivity(CS'), initializeNbTwo(CS'), N, 0, C, N - 1,
  emptyCSL, 10 > if CS' := C,CS ∧ N := cssize(CS') .
```

Hemos decidido mantener el contador de cláusulas de backjump independiente para facilitar la declaración de la estrategia `decide-if-restart`, que estudiaremos a continuación.

Trabajar con tantas heurísticas combinadas tiene la desventaja de que necesitamos mantener actualizada la información de todas ellas. La idea más fácil para definir funciones que actualicen la información en Maude consiste en definir una ecuación independiente por cada una de las estructuras que necesitemos cambiar, asumiendo que tenemos toda la información para el cambio. Posteriormente, podremos declarar operadores con declaraciones más complejas que actualicen toda la información de una vez, incluyendo las primeras declaraciones como condiciones con ecuaciones de matching.

Por ejemplo, a la hora de aprender una cláusula, tenemos que modificar las puntuaciones de todos los contadores para reflejar este nuevo cambio, además de volver a hallar cuál es la cláusula más reciente no satisfecha. Para ello definimos las funciones `updateVarActivity`, `updateLitActivity`, `updateClauseActivity` y `obtainCurrentTopClause` de manera independiente, que luego se agrupan en la siguiente ecuación:

```
ceq updateBerkminInfoForBackjumping(CTX, WCS, CIS, ConflictClause, BackjumpClause, BI)
  = < NewVarActivity, NewLitActivity, NbTwo, NInitial, NewNLearnt, NewTopClause,
  NewActualIdx, NewClauseActivity, NActivityThreshold > if < VarActivity,
  LitActivity, NbTwo, NInitial, NLearnt, TopClause, ActualClauseIdx,
  ClauseActivity, NActivityThreshold > := BI ∧ NewNLearnt := NLearnt + 1 ∧
  ChangeTopClauseInfo := obtainCurrentTopClause(CTX, WCS, ( NInitial + NewNLearnt ) - 1) ∧
  NewTopClause := 1st(ChangeTopClauseInfo) ∧ NewActualIdx := 2nd(ChangeTopClauseInfo) ∧
  CS := obtainIntermediateClausesBackjumping(CIS, ConflictClause, BackjumpClause) ∧
  NewVarActivity := updateVarActivity(VarActivity, (ConflictClause, CS)) ∧
  NewLitActivity := updateLitActivity(LitActivity, BackjumpClause) ∧
  NewClauseActivity := (BackjumpClause :> 0 , updateClauseActivity(CS, ClauseActivity)) .
```

Este razonamiento lo podemos aplicar con cada paso en el que necesitemos acceder o modificar la información contenida en `BerkminInfo`, de manera que a nivel de estrategias la actualización de esta información se haga de la forma más sencilla posible.

A la hora de declarar las estrategias necesarias para simular el nuevo resolutor, recomendamos seguir el esquema de la figura 4.1, pues recoge esencialmente las declaraciones comunes a todos los resolutores que implementan el esquema CDCL. El flujo de llamadas varía dependiendo del resolutor concreto que estemos considerando, aunque se asume que estas modificaciones van a ser mínimas.

Por ejemplo, en el caso de las estrategias del módulo `BERKMIN-STRATEGY`, se incluye una nueva estrategia que se llama desde `fail-branch` para decidir si llevar a cabo o no el proceso de borrado y reinicio:

```
sd decide-if-restart(l, CIS, BI, 150) := WRestart ; matchrew WatchedSeq
  s.t. (M || WCS) := WatchedSeq ∧ RestartInfo := restartTree(BI, M, WCS) ∧
  WCS' := 1st(RestartInfo) ∧ BI' := 2nd(RestartInfo) ∧
  CIS' := generateConflictInfoSetFromBackjump(M, CIS, l) by WatchedSeq using
  ( WChangeWatchedClauseSet[WCS' <- WCS'] ; decide-literal(CIS', BI', 1) ) .

csd decide-if-restart(l, CIS, BI, NRestart) :=
  select-clauses(l, CIS, BI, NRestart + 1) if NRestart < 150 .
```

En este caso vemos por qué hemos decidido dejar fuera el número de cláusulas aprendidas desde el último reinicio y borrado. Hacemos distinción de casos con este número, separando si hemos alcanzado el número de cláusulas aprendidas para realizar este proceso o no.

Hemos intentado sin éxito generar un módulo de estrategias parametrizable que nos sirviese para cualquier implementación. Esto se debe a que el flujo de llamadas entre estrategias varía en función del resolutor entre ellos, lo que dificulta poder generalizar una estructura de módulo.

Por ejemplo, si implementamos Chaff, sabemos que la actualización de la heurística VSIDS se hace según el número de decisiones que hayamos hecho hasta el momento, tal y como hemos estudiado en la sección 5.2. Sin embargo, en el caso de Berkmin, las heurísticas de decisión se actualizan cuando aprendemos nuevas cláusulas.

Capítulo 7

Experimentos

En esta memoria se han incluido 6 estrategias distintas: `classic-dpll-strat`, `basic-dpll-strat`, `watch-literal-dpll-strat`, `jw-heuristic-strat`, `vsids-heuristic-strat` y `berkmin-strat`. Las 3 primeras representan la evolución del algoritmo DPLL al incluir aprendizaje de cláusulas, backjump y el esquema *watch-literal*. Las 3 siguientes incluyen heurísticas basadas en puntuación para elegir literales, incluyendo la estrategia `berkmin-strat` también otras heurísticas para eliminación de cláusulas y reinicios selectivos.

Nuestro objetivo en este capítulo es analizar el impacto que tiene cada una de estas aproximaciones, estudiando qué tamaño de problemas son capaces de asumir y la comparativa de tiempos entre ellos. Así, nos haremos una idea también de la comparación de tiempos entre nuestra implementación en Maude y otros resolutores completos. Todo el código que se ha utilizado para llevar a cabo estos experimentos se puede encontrar en el repositorio https://github.com/alexcere/SAT-Maude.

A la hora de diseñar los experimentos, hemos utilizado instancias uniformemente aleatorias del problema de 3-SAT, distinguiendo según el número de variables que hay en la fórmula y si el problema es satisfactible o no. Estos problemas se suelen utilizar en competiciones de SAT como [10], pues son problemas especialmente difíciles de resolver que ponen a prueba la capacidad de los resolutores. Además, estos problemas tienen una propiedad muy interesante: la probabilidad de que el problema sea satisfactible o no se estabiliza cuando tomamos un número de cláusulas cercano a 4.26 veces el número de variables que estemos considerando para generar el problema. Si la proporción es mayor, la probabilidad de que el problema sea satisfactible disminuye casi a 0, y también se da el caso contrario. Por lo tanto, tenemos una medida certera para generar instancias entre SAT y UNSAT del mismo tamaño.

Estas instancias las hemos sacado de la página web [12], sacando distintas instancias de varios archivos distintos. Estas instancias se corresponden con archivos .cnf escritos en formato DIMACS [11]. Este formato contiene una primera línea indicando el número de variables m y el número de cláusulas del problema n. A continuación, le siguen n líneas que contienen uno o más números en el intervalo $[-m, m]$, representando los números $[1, m]$ literales positivos, $[-m, -1]$ las respectivas negaciones, y el número 0 indica el fin de cláusula.

Este formato no se corresponde con el formato que utilizamos en Maude, así que hemos implementado un programa en C++ que hace la transformación entre ambos formatos. En particular, este programa genera un archivo de texto que incluye las siguientes declaraciones para Maude

```
load <ARGV[1]>
dsrew [1] in <ARGV[2]> : (Problema codificado en Maude) using <ARGV[3]> .
```

donde `ARGV` representa los argumentos de entrada del programa. `ARGV[1]` corresponde al nombre del archivo que contiene las estrategias; `ARGV[2]`, al nombre del módulo de estrategias de este archivo; y `ARGV[3]`, a la estrategia concreta con la que queremos realizar el proceso de reescritura.

Así, hemos generado también scripts en bash que recorren cada instancia del problema en su versión .cnf, obtienen las correspondientes llamadas que se necesitan para ejecutar en Maude y redirigen este flujo de entrada al motor de reescritura de Maude, para que lo ejecute. Por último, redirigen la salida de Maude a un fichero independiente, que contiene la solución al problema, el número de reescrituras y el tiempo que ha tardado Maude en llegar a esta solución.

Hemos dividido los problemas en cuatro categorías en función del número de variables que intervienen en ellas: fácil, medio, difícil y muy difícil. Cada una de estas categorías contiene 30 instancias, siendo 15 de ellas satisfactibles y 15 insatisfactibles.

La categoría fácil considera instancias satisfactibles de 20, 50, 75, 100 y 125 variables, mientras que las instancias insatisfactibles son de 50, 75, 100 y 125 variables. En el caso de la categoría medio, en

Tipo Instancia	Nº instancias	classic	basic	watch-literal	jw	vsids	berkmin
SAT-Fácil	15	1	9	10	14	15	14
UNSAT-Fácil	15	0	3	7	9	9	11
SAT-Medio	15	0	0	0	4	5	4
UNSAT-Medio	15	0	0	0	0	0	0
Total	60	1	12	17	27	29	29

Figura 7.1: Comparativa de archivos analizados correctamente por cada estrategia

ambos casos consideramos instancias que oscilan desde los 150 literales hasta los 250, con un paso de 25 variables entre distinto tipo de instancias. La categoría difícil considera instancias desde 275 hasta 375 variables, mientras que la categoría muy difícil incluye instancias entre 400 y 500 variables. Con esta distribución, tenemos 3 o 4 instancias de cada tipo de problema, analizando un total de 120 archivos. Para cada instancia de estas categorías, hemos incluido un *timeout* de 30 minutos tras el cual se abortaba la ejecución.

Lamentablemente, nuestras estrategias solo han podido asumir instancias de las categorías fácil y medio, así que a partir de este punto obviamos las otras categorías.

Para comprobar que efectivamente las soluciones obtenidas en estos experimentos se corresponden con asignaciones válidas en el caso de ser satisfactibles, hemos declarado el módulo funcional `TESTER`, que contiene la operación `reduceBasicSequent`

```
op reduceBasicSequentAuxiliar : BasicSequent ClauseSet → Bool .

eq reduceBasicSequentAuxiliar(M || emptyCS, CS') = true .
ceq reduceBasicSequentAuxiliar((M || (l \/ C) , CS), CS') =
  reduceBasicSequentAuxiliar((M || CS), CS') if (~(l) in CS') = false .
ceq reduceBasicSequentAuxiliar((M || C , CS), CS') =
  reduceBasicSequentAuxiliar((M || CS), CS') if M |= C .
eq reduceBasicSequentAuxiliar(BasicSeq, CS) = false [owise] .

op reduceBasicSequent : BasicSequent → Bool .

eq reduceBasicSequent(M || CS) = reduceBasicSequentAuxiliar((M || CS), CS) .
```

La idea de esta operación es sencilla: eliminar todas las cláusulas que se han obviado debido a la aplicación de la regla `WUnitPropagate` o que son satisfechas por el contexto actual. Si al final obtenemos la fórmula vacía, entonces la asignación propuesta es válida.

Los experimentos se han llevado a cabo en un procesador Common KVM processor de 2.095 GHz x 4 y de 4.04 GB de memoria, que corre Debian 4.19. Para ello, hemos utilizado la versión de core Maude 3.0 para Linux64.

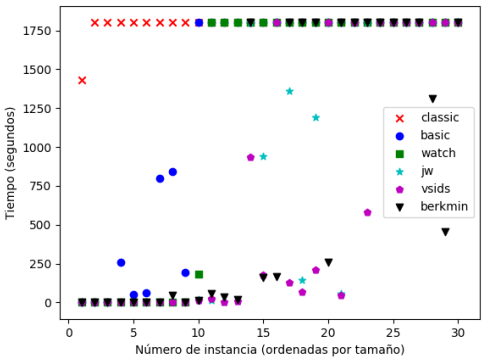
En la figura 7.1, se muestra una comparativa entre el número de archivos que se han podido analizar correctamente utilizando cada estrategia, teniendo en cuenta el *timeout* de 30 min.

La estrategia `classic` es mucho más ineficiente que el resto de estrategias, siendo únicamente capaz de analizar correctamente uno de los archivos. Esto demuestra que la utilización de la regla de `Backjump` tiene un impacto muy grande a la hora de abordar el problema del SAT.

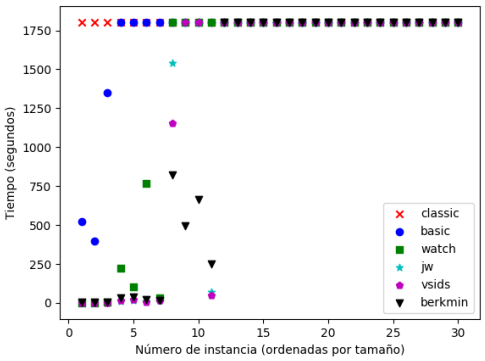
También es interesante ver la diferencia entre las estrategias `basic` y `watch-literal`. Utilizando este esquema, hemos conseguido analizar 5 instancias más correctamente que las analizadas por el algoritmo DPLL con aprendizaje, lo que indica que este planteamiento reduce considerablemente el tiempo de ejecución. Considerando que en el problema del 3-SAT no nos ahorramos tantas comparaciones cuando intentamos aplicar la regla del `UnitPropagate`, creemos que gran parte de la mejora radica en tener que evitar comparar todas las cláusulas para detectar si hemos llegado a conflicto.

Las estrategias `watch-literal` y `jw` utilizan literalmente las mismas declaraciones, salvo que en el caso de `jw`, precomputamos su heurística. Por tanto, deducimos que el uso de heurísticas para la toma de decisiones importa mucho a la hora de mejorar la eficiencia, pues de esta manera, siempre podemos explorar caminos más prometedores.

Vemos que las 3 últimas estrategias consiguen analizar esencialmente el mismo número de instancias. Destaca `vsids` a la hora de analizar instancias de SAT, mientras que `berkmin` consigue despuntar en el caso de instancias de UNSAT. Este hecho parece indicar que a priori `vsids` consigue dar con caminos suficientemente buenos con sus heurísticas, pero en caso de que un camino prometedor resulte no contener una asignación válida, entonces no es capaz de abandonarlo. La estrategia `berkmin` puede abordar esta



(a) Tiempos de ejecución por cada estrategia para cada problema satisfactible analizado



(b) Tiempos de ejecución por cada estrategia para cada problema insatistactible analizado

Figura 7.2: Comparativa de tiempos de ejecución entre las distintas estrategias

problemática con los reinicios selectivos y eliminación de cláusulas, aunque no existe mucha diferencia en la tasa de acierto con respecto a `vsids`.

Creemos que este comportamiento se debe a que somos demasiado poco restrictivos con el borrado de cláusulas para las instancias que estamos considerando. Teniendo en cuenta que las cláusulas son de tamaño 3 y las instancias no contienen un número elevado de variables, ocurre que es difícil obtener cláusulas de backjump de tamaño mayor que 7, por lo que a la hora de borrar cláusulas antiguas, solamente unas pocas son borradas. El proceso se termina volviendo demasiado ineficiente.

También queremos destacar que los mecanismos propuestos para `berkmin` a la hora de equilibrar la exploración de ramas parecen haber dado frutos, pues esta estrategia es la que más instancias de problemas insatisfactibles es capaz de resolver. Este equilibrio es muy importante en estos casos, pues para detectar UNSAT, tenemos que terminar cortando todas las ramas del árbol de derivación.

Para tener una visión global de la comparativa entre unas estrategias y otras, hemos incluido en la figura 7.2 una comparativa entre los tiempos que se han tardado en ejecutar cada uno de los casos del experimento. Hemos decidido dividir entre problemas satisfactibles, en la figura 7.2a, y no satisfactibles, en la figura 7.2b.

Lo primero que nos llama la atención de la figura 7.2 es que la mayoría de tiempos de las primeras 15 instancias son bastante bajas para todos los resolutores, mientras que en el resto de instancias, los tiempos son por lo general bastante más altos. Las estrategias `classic` y `basic` siempre son mucho peores en comparación al resto, mientras que las otras 4 tienen tiempos parecidos en el resto de casos.

Por lo general, `watch-literal` se empieza también a quedar atrás conforme aumentamos el número de variables. Las estrategias restantes empiezan a mostrar diferencias mayores conforme aumentamos el número de literales a considerar: la estrategia `jw` suele quedar siempre por detrás de la estrategia `vsids`, y esta también suele ser más rápida que `berkmin`. Sin embargo, existen varias instancias en las que `berkmin` consigue encontrar asignaciones válidas, mientras que ninguna de las otras dos son capaces. Esto es especialmente reseñable con los últimos problemas, que corresponden a instancias con 225 y 250 literales, siendo `berkmin` capaz de tratar con ellos.

En el caso de los problemas insatisfactibles, nuestras estrategias no son capaces de asumir la mayoría de las instancias. Esto se debe a que los problemas insatisfactibles son más difíciles de probar en resolutores completos, y el uso de heurísticas para tomar decisiones no es tan relevante. En estos casos, es más importante gestionar correctamente las cláusulas aprendidas y reiniciar para evitar caminos infructuosos, equilibrando a su vez las búsquedas en distintos árboles. Por ello `berkmin` destaca sobre el resto de estrategias, pues adopta mecanismos para tratar con todos estos problemas.

Conforme aumentamos el tamaño de las instancias, los tiempos aumentan drásticamente. Esto lo podemos ver con las instancias de 100 y 125 literales, que son muchísisimo más costosas que las de 50 y 75 literales. En general, los problemas insatisfactibles son más difíciles de resolver y llevan las capacidades del resolutor al límite.

Para concluir, queremos destacar que estos tiempos no pueden competir con los resolutores modernos, así que no es recomendable su uso para cuestiones prácticas. Sin embargo, teniendo en cuenta que Maude es un lenguaje de alto nivel, nuestras estrategias han probado ser capaces de asumir instancias

relativamente grandes para este tipo de lenguajes.

Capítulo 8

Conclusiones

Nuestra idea inicial con este proyecto era plantear un sistema complejo en el que estudiar la aplicación del nivel de estrategias de Maude. Hemos conseguido alcanzar este objetivo, y además, llevar nuestro planteamiento a otros niveles igual de relevantes. Vamos a recoger en este capítulo brevemente cuáles son los aspectos fundamentales que hemos conseguido transmitir con esta memoria.

El primer punto que queremos destacar es el tratamiento detallado que se ha hecho de las distintas técnicas que utilizan los resolutores actuales para abordar el problema del SAT. Este problema ha sido estudiado en detalle en distintos trabajos, pero la mayoría de estos se centran en aspectos muy puntuales del mismo. La bibliografía más accesible a nivel de lectores que se quieren iniciar en el problema del SAT se basa en transparencias donde se introducen estas técnicas de forma muy esquemática, obviando la parte teórica y de implementación del mismo. Este trabajo ha intentado aunar explicación, teoría e implementación de forma que sea accesible y útil para todo aquel que se quiera iniciar en este tema.

De nuevo, no podemos dejar de destacar la importancia del trabajo de Tinelli [9], que ha supuesto un importante punto de partida y una inspiración a la hora de abordar este tema. Consideramos que esta memoria complementa a la perfección este trabajo, pues hemos implementado de forma directa los sistemas de reglas planteados; puntualizando y formalizando algunas de los aspectos que podían quedar sueltos en ciertos planteamientos de este trabajo. Por ejemplo, se ha incluido las declaraciones formales de algunos predicados que son necesarios para la declaración de reglas, y que en el trabajo de Tinelli se incluían de forma más coloquial.

Esto nos lleva precisamente al siguiente punto de este trabajo. Esta memoria constituye el culmen de la primera incursión del autor en dos campos de investigación muy extensos: el del problema del SAT y el de la lógica de escritura. Como tal, la propia estructura del texto, los temas que se han elegido para tratar y la forma en la que se presentan muestran su visión particular de ambos mundos, después de haberse formado durante meses con distintas fuentes.

En particular, creemos que esto se hace explícito en el capítulo 4, en el cual conseguimos llevar a un nuevo nivel el conocimiento del problema de SAT y de Maude. Por un lado, hemos propuesto un sistema de reglas totalmente novedoso a partir de una técnica considerada de bajo nivel, abstrayendo las ideas fundamentales de la misma y formalizando su planteamiento. Por otro lado, en este capítulo se consigue fusionar los niveles de reglas y estrategias de forma razonada, al repartir condiciones de reglas entre ambos niveles para sacar el máximo rendimiento.

Además, hemos comprobado en el capítulo 7 que nuestro sistema de reglas propuesto supera al del algoritmo DPLL básico, lo cuál supone que nuestro sistema de inferencia es eficiente.

Otro punto que consideramos clave en esta memoria es el estudio pormenorizado que se hace de la implementación. Hemos obviado deliberadamente aquellas declaraciones que no aportan demasiado a nivel de entender los fundamentos teóricos, centrándonos así sobre todo en el nivel de reglas y estrategias. No obstante, también hemos incluido la declaración de las ecuaciones que son fundamentales a la hora de realizar la implementación, para que el lector disponga de las herramientas necesarias para desarrollar sus propias extensiones de otros resolutores SAT teniendo un trabajo previo sólido. Estas explicaciones se pueden generalizar a planteamientos y problemas que nos podemos encontrar con otros sistemas que queramos implementar en Maude.

Por último, queremos mencionar la introducción al lenguaje Maude del apéndice A. Este apéndice condensa en unas pocas páginas los fundamentos necesarios para empezar a programar en Maude, introduciendo su sintaxis, el funcionamiento de los distintos módulos y ejemplos útiles que complementan este conocimiento. Obviamente, el manual de Maude [2] consigue este propósito con mucho más detalle

y mimo, pero consideramos que este capítulo puede ilustrar de forma muy accesible a aquellos usuarios que quieren entender de forma más superficial las posibilidades de este lenguaje.

La parte experimental no es uno de los puntos fuertes del trabajo, aunque sí lo consideramos relevante para hacer una comparativa entre distintas estrategias y comprobar que funcionan aunque no se puedan aplicar a grandes casos. También nos puede servir de cara a estudiar configuraciones óptimas de parámetros en ciertas heurísticas.

Este trabajo es bastante completo tal y como está planteado. Se ha hecho una revisión exhaustiva de las técnicas más comunes a todos los resolutores SAT, así como introducir algunas heurísticas muy conocidas. El capítulo 6, en el que se introduce el resolutor Berkmin, cierra con los objetivos que habíamos perseguido: conseguir una implementación completa en Maude y abarcar todos los posibles aspectos a la hora de construir un resolutor.

La gran cantidad de técnicas y heurísticas existentes nos permiten seguir extendiendo el trabajo, aunque creemos que estas incorporaciones estarían demasiado centradas en aspectos concretos de los resolutores y perderíamos nuestra motivación inicial de introducir de forma clara el tema.

Bibliografía

[1] Armin Biere. PicoSAT Essentials. *JSAT*, 4:75–97, 05 2008. doi:10.3233/SAT190039.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, y Carolyn Talcott. Maude manual (version 3.0). *SRI International–University of Illinois at Urbana-Champaign*, 2019.

[3] Steven Eker, Narciso Martí-Oliet, José Meseguer, Isabel Pita, Rubén Rubio, y Alberto Verdejo. Strategy language for Maude. http://maude.sip.ucm.es/strategies/, 2020. [Online; acceso 28-6-2020].

[4] Niklas Eén y Niklas Sörensson. An Extensible SAT-solver. En Enrico Giunchiglia y Armando Tacchella, editores, *SAT*, tomo 2919 de *Lecture Notes in Computer Science*, páginas 502–518. Springer, 2003.

[5] Eugene Goldberg y Yakov Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549 – 1561, 2007. ISSN 0166-218X. doi:https://doi.org/10.1016/j.dam.2006.10.007. SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[6] Robert G Jeroslow y Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[7] José Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012. ISSN 1567-8326. doi:https://doi.org/10.1016/j.jlap.2012.06.003. Rewriting Logic and its Applications.

[8] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, y Sharad Malik. Chaff: Engineering an Efficient SAT Solver. En *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, páginas 530–535. Association for Computing Machinery, New York, NY, USA, 2001. ISBN 1581132972. doi:10.1145/378239.379017.

[9] Robert Nieuwenhuis, Albert Oliveras, y Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, 53(6):937–977, noviembre 2006. ISSN 0004-5411. doi:10.1145/1217856.1217859.

[10] The International SAT Competition Web Page. http://www.satcompetition.org/, 2002. [Online; acceso 28-6-2020].

[11] SAT Competition 2009: Benchmark Submission Guidelines. http://www.satcompetition.org/2009/format-benchmarks2009.html, 2019. [Online; acceso 28-6-2020].

[12] SATLIB-Benchmark Problems. https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html, 2011. [Online; acceso 28-6-2020].

[13] Cesare Tinelli. A DPLL-Based Calculus for Ground Satisfiability Modulo Theories. En *Proceedings of the European Conference on Logics in Artificial Intelligence*, JELIA '02, páginas 308–319. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3540441905.

Apéndice A

El lenguaje Maude en detalle

El enfoque que haremos de este apéndice será presentar las funcionalidades necesarias para entender la implementación del proyecto, dejando fuera deliberadamente otras funcionalidades que no han sido relevantes.

Consideramos que con esta información el lector puede tener una visión más o menos global de qué se puede hacer en Maude y cómo hacerlo. Para más detalle, se recomienda consultar directamente la documentación oficial de Maude 3.0 [2].

A.1. Sintaxis básica

En este apartado, analizaremos las distintas funcionalidades básicas de Maude a nivel de sintaxis. Para ello, utilizaremos distintos ejemplos sacados del propio código del proyecto, para así poner en relieve la importancia que tiene entender cómo funciona Maude para sacarle el máximo partido. En este apartado, los ejemplos han sido recogidos del módulo `LOGIC`.

A.1.1. Tipado y *kind*

Los tipos se declaran utilizando la palabra reservada `sort` seguida de un espacio y un punto:

```
sort <Sort> .
```

En el caso de querer declarar varios tipos a la vez, podemos utilizar `sorts`, seguido de todos los tipos que queramos declarar:

```
sorts <Sort-1> <Sort-2> ... <Sort-n> .
```

También se pueden declarar tipos estructurados, que consisten en tipos cuya declaración viene determinada por otros tipos. Esta declaración viene dada por el nombre del tipo que queremos declarar, seguido de la lista de tipos que queremos instanciar entre llaves. Así, por ejemplo, en el módulo `PAIR`, declaramos el tipo `Pair` de la siguiente forma:

```
sorts Pair{X,Y} .
```

donde `X` e `Y` represetan dos módulos cualesquiera. Hablaremos más detenidamente sobre la parametrización de módulos en la sección A.4.2.

Se pueden definir relaciones de inclusión entre distintos tipos, utilizando la palabra reservada `subsort`. Para entender cómo funciona, vamos a recurrir a la definición básica de tipos del módulo `LOGIC`:

```
sorts Literal Context Clause ClauseSet .
subsorts Qid < Literal < Context Clause < ClauseSet .
```

En esta definición, el tipo `Qid` representa un tipo básico de Maude, que se utiliza para representar identificadores genéricos. Estos identificadores consisten en una serie de caracteres precedidos por el caracter '.

La jerarquía descrita anteriormente indica que cualquier identificador `Qid` puede entenderse a su vez como un elemento del tipo `Literal`. Ahora bien, en la siguiente relación aparecen dos elementos en vez de uno, lo que quiere indicar que el tipo `Literal` es subtipo a la vez de `Context` y `Clause`, no existiendo relación de inclusión entre ambos tipos.

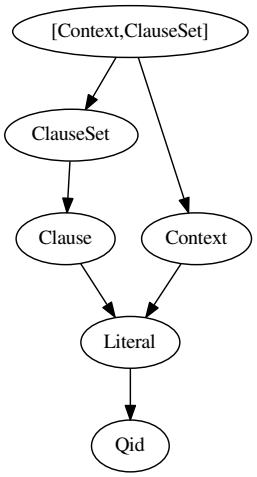


Figura A.1: Jerarquía de los tipos definidos en el módulo `LOGIC`, incluyendo el *kind*

Por tanto, esta jerarquía de tipos induce un orden parcial entre ellos, de tal manera que el conjunto de tipos con la relación de inclusión constituye un conjunto parcialmente ordenado.

En esta situación, es donde cobra relevancia el concepto de *kind*. Un conjunto parcialmente ordenado puede tener varios elementos maximales, de tal manera que entre ellos no se pueda aplicar la relación de inclusión. El *kind* de una jerarquía se entiende como un supertipo que engloba a todos los tipos. Una expresión pertenecerá al *kind* cuando no se consiga reducir completamente al tipo al que estábamos intentando reducir.

Por tanto, el *kind* se entiende como un supertipo de error, de tal manera que el usuario puede manejarlo a su antojo. Esto nos permite simplificar la declaración de algunas ecuaciones en ciertos casos, e incluso podemos definir operaciones en este nivel.

También hay que tener en cuenta que este concepto es relevante a nivel de reglas, pues solo podremos declarar transiciones entre tipos que compartan el mismo *kind*.

El *kind* viene dado por el nombre de uno o varios de los tipos de la jerarquía entre corchetes. En su forma canónica, viene dado por los elementos maximales del conjunto separados por coma.

La figura A.1 muestra la jerarquía de tipos completa, incluyendo el *kind*.

A.1.2. Operadores y atributos de operadores

Una vez hemos analizado cómo funcionan los tipos, podemos pasar a cómo se definen los operadores en Maude. La declaración de operadores se hace siguiendo la siguiente expresión:

```
op <OpName> : <Sort-1> ... <Sort-k> → <Sort> [<OperatorAttributes>] .
```

donde los primeros k tipos representan el dominio del operador, y el número k representa la aridad. Nótese que es posible que $k = 0$, lo cual significa que el operador está definiendo una constante.

Los operadores por defecto están declarados de forma prefija. Si incluimos un número de guiones bajos (_) igual a la aridad del operando, entonces estaremos declarando el operador en forma infija. Así, por ejemplo, tenemos los siguientes operadores declarados para el tipo `Context`:

```
op emptyCTX : → Context .
op __ : Context Context → Context [ctor assoc id: emptyCTX prec 30] .
op _in_ : Literal Context → [Bool] .
```

Nótese que el segundo operador está declarado en forma infija (con notación vacía), de tal forma que dos elementos de la clase `Context` separados por espacio se encuentran agrupados por este operador.

En este mismo operador, podemos ver la declaración de algunos atributos. Vamos a explicar todos aquellos que hemos utilizado en nuestra implementación:

- `assoc`: indica que el operador es asociativo.

- `comm` : indica que el operador es conmutativo.
- `id:` : se especifica cuál es el elemento identidad. Esta especificación es muy útil para el encaje de patrones con operadores asociativos.
- `ctor`: indica que el operador es un constructor, diferenciándolo así de otros operadores cuyo objetivo es reducir una expresión a una forma canónica. En nuestro caso, no tiene mucha relevancia, pero puede tenerla para la depuración y demostración de teoremas.
- `prec`: es un atributo que viene seguido de un número, e indica la precedencia del operador. Es decir, al analizar un término, los operadores que aparecen en él tienen una precedencia definida, de tal forma que se asocian según esos valores. Maude tiene predefinida la precedencia por defecto según el tipo de operador.

A.1.3. Variables

Las variables representan un valor cualquiera de un tipo o de un *kind* concreto. Se utilizan para expresar de forma general conjuntos de valores de un cierto tipo a la hora de declarar ecuaciones, reglas o estrategias.

Se pueden declarar directamente dentro de una expresión o de forma global dentro de un módulo. En nuestro caso, todas las declaraciones de variables son globales y se especifican con la siguiente sintaxis:

```
var p : Qid .
vars l u : Literal .
```

A.2. Módulos funcionales

En Maude los módulos funcionales se declaran utilizando las palabras reservadas:

```
fmod <ModuleName> is <DeclarationsAndStatements> endfm
```

En la sección de *Declaraciones*, se pueden incluir tanto declaraciones de operadores, como ecuaciones y relaciones de pertenencia. Estas últimas no van a aparecer en nuestro código, así que nos vamos a centrar directamente en las ecuaciones. Estas pueden ser de dos tipos: incondicionales o condicionales.

Las ecuaciones incondicionales son ecuaciones del tipo $t = t'$, que cumplen que t y t' están en el mismo kind y que el tipo de t' es un subtipo del tipo de t de acuerdo a la jerarquía de tipos. Se declaran con la siguiente sintaxis:

```
eq <Term-1> = <Term-2> [<StatementAttributes>] .
```

Así, por ejemplo, si estudiamos las ecuaciones correspondientes al operador `ctxsize`, tendríamos las siguientes ecuaciones:

```
eq ctxsize(emptyCTX) = 0 .
eq ctxsize(M l) = 1 + ctxsize(M) .
```

Nótese que en estas ecuaciones se combina el encaje de patrones con ecuaciones recursivas. En este caso, podemos ver que el término `M l` corresponde a aplicar el operador `__` definido anteriormente a las variables `M` y `l`.

Por otro lado, las ecuaciones condicionales son de la forma $t = t'$ *if* $C_1 \wedge ... \wedge C_n$, pudiendo ser las condiciones $C_1, ..., C_n$ de tres tipos:

- Ecuaciones del tipo $t = t'$, como las descritas anteriormente. En este caso, se cumplirá la ecuación si al reducir ambos términos llegamos al mismo término canónico.
- Ecuaciones de *matching*, de la forma $t := t'$. Una condición de este tipo tiene éxito si se puede encajar el término de la izquierda con el de la derecha. En nuestro caso, recurriremos a estas ecuaciones para comprobar si se ha podido reducir o no un término completamente.
- Ecuaciones booleanas abreviadas, que consisten en expresiones cuyo término corresponde a un término en `[Bool]`. Estas expresiones se corresponden con ecuaciones del tipo $t == true$, en las que se omite el predicado $==$.

En términos matemáticos, el orden de evaluación de las condiciones no es relevante, pero sí lo es a nivel operacional. Su correspondiente forma de codificarlo en Maude es la siguiente:

```
ceq <Term-1> = <Term-2> if <EqCondition-1> ∧ ... ∧ <EqCondition-k> [<StatementAttributes>] .
```

La definición de un operador puede combinar ecuaciones incondicionales con condicionales. Podemos verlo en el siguiente ejemplo:

```
ceq obtainBackjumpClause(CIS, C) = C if literalsInLevel(CIS,C) == 1 .
ceq obtainBackjumpClause(CIS l → (C \/ l) , ~(l) \/ D) =
  obtainBackjumpClause(CIS, C \/ D) if literalsInLevel(CIS l → (C \/ l), ~(l) \/ D) =/= 1 .
ceq obtainBackjumpClause(CIS ~(l) → (C \/ ~(l)), l \/ D) =
  obtainBackjumpClause(CIS, C \/ D)
  if literalsInLevel(CIS ~(l) → (C \/ ~(l)), l \/ D) =/= 1 .
eq obtainBackjumpClause(CIS l → C, D) = obtainBackjumpClause(CIS, D) [owise] .
```

En este ejemplo, se combinan ecuaciones condicionales e incondicionales para definir la semántica del operador `obtainBackjumpClause`. La última ecuación es ligeramente distinta de las otras. En ella, se ha incluido el atributo `owise`. A grandes rasgos, este atributo indica que la última ecuación cubre todos los casos que no han sido satisfechos por las ecuaciones anteriores.

Ahora cabe preguntarse si cualquier declaración de ecuación tanto condicional como incondicional escrita en Maude puede aplicarse en su motor de reescritura. Para que Maude pueda evaluar las ecuaciones correctamente, estas deben cumplir ciertas condiciones: las condiciones de *Church-Rosser* y de terminación, así como las condiciones de admisibilidad. La primera condición asegura que el resultado de la aplicación de varias ecuaciones es independiente del orden en el que se apliquen, mientras que la segunda condición indica que eventualmente se llega a un término que está bien definido en un tipo concreto o que pertenece al *kind* asociado. Con estas condiciones, tiene sentido hablar de términos canónicos, que son aquellos que no pueden simplificarse utilizando ecuaciones.

Las condiciones de admisibilidad, por otro lado, indican qué condiciones adicionales hay que exigirle a cualquier ecuación para que pueda ser ejecutada por el motor de Maude. Estas tienen que ver con la relación entre variables en el lado izquierdo y derecho de una ecuación. Por ejemplo, si en el lado derecho aparece una variable que no ha sido declarada previamente en el lado izquierdo o en una ecuación de matching, entonces la ecuación correspondiente no es admisible.

Las condiciones referidas anteriormente son también aplicables al nivel de reglas, aunque en el caso de la admisibilidad, hay ligeros cambios en su definición.

Maude permite declarar ecuaciones no admisibles si incluimos el atributo `nonexec`. A priori, puede parecer poco útil declarar ecuaciones o reglas que no puedan ser aplicadas por el motor de reescritura de Maude. Sin embargo, existen ciertas situaciones en las que es necesario utilizar esta declaración para obtener comportamientos que serían muy complicados de simular de otra manera.

Para poner un ejemplo de ello, vamos a comentar brevemente la regla *Backjump* de las ecuaciones de Tinelli. Con esta regla, se aprende una nueva cláusula que hemos generado a partir de un conflicto. No obstante, una cláusula de aprendizaje se puede generar de distintas formas, por lo que parece lógico que de alguna manera se haga una separación entre el proceso de generación y el aprendizaje en sí. Esto se logra declarando la regla como no ejecutable:

```
rl [Learn] : M || CS ⇒ M || CS,C [nonexec] .
```

A nivel de estrategias, podremos instanciar la regla con una clausula concreta, de manera que a este nivel es donde podemos elegir el enfoque que queramos para generarla.

A.3. Módulos de sistema

En Maude los módulos de sistema se declaran utilizando la siguiente sintaxis:

```
mod <ModuleName> is <DeclarationsAndStatements> endm
```

Como hemos indicado en la introducción del capítulo, en la sección de *Declaraciones* podemos incluir todas las funcionalidades comentadas en la sección A.2, así como la declaración de reglas.

Una regla representa transiciones no deterministas entre términos. Al proceso de aplicación de reglas se le conoce como la reescritura de términos, y en principio no tenemos forma de controlar cómo se realiza. Existen ciertos comandos para llevar a cabo la reescritura de términos de distintas formas, pero no los vamos a tratar, ya que en nuestro caso únicamente manejaremos estrategias.

De nuevo nos encontramos con dos tipos de reglas: las reglas condicionales e incondicionales. Vamos a empezar estudiando las reglas incondicionales, pues son más sencillas de entender.

La expresión de una regla es de la forma $l : t \Rightarrow t'$, siendo l la etiqueta de la regla, y t, t' dos términos que pertenecen al mismo *kind*. En Maude, se especifican con la siguiente sintaxis:

```
rl [<Label>] : <Term-1> ⇒ <Term-2> [<StatementAttributes>] .
```

Ya vimos en la sección anterior un ejemplo de regla, en el que además se utilizaba el atributo `nonexec`. En el caso de las reglas, el atributo `owise` no se puede aplicar, ya que las reglas son independientes entre sí y no tiene sentido especificar ese comportamiento.

En el caso de las reglas condicionales, su expresión matemática es $l : t \Rightarrow t'$ *if* $C_1 \wedge ... \wedge C_n$, donde $C_1, ..., C_n$ corresponden a las condiciones que se imponen para aplicar la regla, y corresponden a las mismas del caso de las ecuaciones condicionales, añadiendo una nueva posibilidad:

- Condiciones del tipo $p \Rightarrow q$. Estas condiciones expresan que existe una forma de reescribir p en q utilizando reglas. Veremos que a nivel de estrategias se puede especificar qué reglas se quieren aplicar a la hora de hacer la comprobación.

La sintaxis de una regla condicional en Maude es la siguiente:

```
crl [<Label>] : <Term-1> ⇒ <Term-2>
if <Condition-1> ∧ ... ∧ <Condition-k>
[<StatementAttributes>] .
```

A.4. Operaciones de módulos

En los apartados anteriores, hemos aprendido cómo se programan los módulos funcionales y de sistema. Maude permite estructurar el código en distintos módulos, de manera que se pueda reutilizar el código, así como dividir los programas en pequeños módulos que sean entendibles para el usuario.

Para entender cómo trabajar a nivel de módulos, primero aprenderemos a importar módulos ya existentes en la sección A.4.1.

Maude también permite definir módulos genéricos que contienen operaciones que deben ser definidas por el módulo concreto que lo instancie. Estudiaremos la creación de teorías y la parametrización de módulos en la sección A.4.2.

A.4.1. Importación de módulos

En Maude, la importanción de módulos es transitiva. Si un módulo incluye un submódulo que a su vez incluye otros, el módulo inicial contendrá a todos los submódulos del árbol de inclusiones. Para hacer una importación de módulos válida, se tiene que verificar que la jerarquía definida por las inclusiones sea acíclica; es decir, que no se pueda dar el caso de que un módulo se incluya como submódulo a él mismo.

En Maude se puede importar cualquier submódulo de tres modos distintos: `protecting`, `extending` o `including`. Estas declararaciones se suelen incluir justo después de declarar el módulo que va a importar, y su sintaxis es de la siguiente forma

```
protecting <ModuleExpression> .
extending <ModuleExpression> .
including <ModuleExpression> .
```

aunque se puede abreviar

```
pr <ModuleExpression> .
ex <ModuleExpression> .
inc <ModuleExpression> .
```

Si importamos un módulo como `protecting`, significa que no vamos a introducir nuevos contructores o constantes que expandan las posibles formas canónicas definidas en alguno de los módulos que vamos a importar, ni vamos a introducir nuevas ecuaciones que entren en conflicto con alguna definida anteriormente. En el caso de `extending`, permitimos introducir nuevos elementos canónicos, pero sin confusión entre definiciones. Por último, `including` no garantiza ninguna de las dos propiedades.

En nuestro caso, hemos estructurado el código de tal manera que todas las importaciones se hacen con `protecting`.

A.4.2. Teorías, vistas y módulos parametrizados

Maude dispone de ciertos mecanismos para poder reutilizar código. Entre ellos destaca la definición de teorías y parametrización de módulos.

Estos mecanismos se asemejan a lo que se entiende en lenguajes de programación usuales al uso de interfaces y clases abstractas, de tal manera que podemos definir esqueletos de implementaciones, asumiendo que tienen definidos ciertos operadores que cumplen ciertas propiedades para luego poder instanciar módulos concretos. En este esquema, la teoría representaría la interfaz donde se definen la estructura y propiedades, la vista correspondería con una implementación concreta de esa interfaz, y la parametrización en un módulo correspondería a la instanciación concreta en una clase que hace uso de la interfaz.

En nuestro caso, vamos a utilizar este esquema de manera muy superficial: únicamente se utiliza en la clase *utils*, donde hemos incluido las declaraciones de los módulos funcionales *PAIR* y *TRIPLE*, que representan pares y tripletas de elementos, respectivamente.

Para ello, utilizaremos la teoría funcional *TRIV*, que define únicamente un tipo genérico:

```
fth TRIV is
  sort Elt .
endfth
```

Para indicar cómo se adapta una teoría a un módulo concreto, es necesario definir una vista, donde se especifique a qué tipo u operación específica del módulo se vincula cada uno de la teoría. Así, por ejemplo, podemos adaptar el tipo *Clause* a la teoría *TRIV* declarando la vista *Clause*:

```
view Clause from TRIV to LOGIC is
  sort Elt to Clause .
endv
```

Nótese que hemos elegido el nombre de la vista para que coincida con el del tipo, para no tener que manejar distintos nombres a la hora de incluir los tipos.

Ahora veamos cómo se declara un módulo parametrizado con el ejemplo del módulo *PAIR*:

```
fmod PAIR{X :: TRIV, Y :: TRIV} is
  sort Pair{X, Y} .
  op <_;_> : X$Elt Y$Elt → Pair{X, Y} .

  op 1st : Pair{X, Y} → X$Elt .
  op 2nd : Pair{X, Y} → Y$Elt .

  var A : X$Elt .
  var B : Y$Elt .

  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .

endfm
```

Vemos que en la declaración del módulo se incluye entre llaves las teorías que se tienen que parametrizar. En este caso, hay que instanciar el módulo *PAIR* con dos vistas que parametricen la teoría *TRIV*.

Para referirnos a los tipos y operaciones de cada uno de los módulos genéricos, usamos el nombre genérico del mismo seguido de `$` y del tipo o de la operación de la teoría a la que nos queramos referir.

Para instanciar el módulo parametrizado, basta incluir el nombre del módulo con el nombre de las vistas concretas entre llaves. Por ejemplo, si queremos importar el módulo *PAIR* con pares de tipos *Nat* y *Clause*, entonces debemos incluir la siguiente importación

```
pr PAIR{Clause, Nat} .
```

de tal manera que ahora disponemos del tipo

```
Pair{Clause, Nat} .
```

y sus operaciones pertinentes, con las que trabajamos como si se tratase de un tipo normal.

A.5. Lenguaje de estrategias

El lenguaje de estrategias de Maude lleva varios años en existencia, pero no ha sido hasta la versión Maude 3.0 cuando se ha incluido de forma oficial con su implementación completa.

Este lenguaje se basa en la combinación de ciertos operadores de estrategias de tal manera que se pueden crear estrategias más complejas. Estos operadores se aplican sobre reglas y otras estrategias, de tal manera que por lo general el término sobre el que se está aplicando la estrategia suele quedar oculto en este nivel, aunque no necesariamente. A este término lo denotaremos como término sujeto.

Las estrategias se pueden aplicar directamente sobre el intérprete de Maude, o dentro de módulos de estrategias. Para ejecutar una estrategia en el intérprete, se pueden utilizar los comandos `srew [n] t using` α o `dsrew [n] t using` α. Estos comandos difieren en el modo que tiene el motor de reescritura de recorrer el árbol de derivación.

Maude realiza el proceso de aplicación de reglas siguiendo una búsqueda en el árbol. Las ramas del árbol de exploración corresponden con las distintas decisiones que el motor de Maude hace al aplicar una regla o estrategia, pues normalmente podemos aplicarlas de distintas formas, generando distintos términos.

Usando el primer comando, la exploración es parecida a una búsqueda en anchura, llevando distintas ramas del árbol de exploración a la vez. De esta manera, se garantiza que eventualmente se alcanzan todas las soluciones si hay suficiente memoria. En el segundo caso, la búsqueda es en profundidad. Puede ocurrir que esta búsqueda no acabe si nos encontramos con ramas infinitas del árbol de derivación.

El parámetro `n` de la expresión anterior corresponde con el número de soluciones que queremos hallar utilizando la estrategia, y el parámetro α corresponde con la estrategia concreta a utilizar.

Una vez sabemos cómo ejecutar estrategias utilizando comandos, vamos a pasar al estudio de las mismas.

A.5.1. Estrategias básicas

Las estrategias más sencillas en Maude son las estrategias `idle` y `fail`. La primera devuelve el término sujeto tal cual, mientras que la segunda no devuelve ningún resultado.

Otra estrategia sencilla consiste en la aplicación de una regla concreta. Su expresión es de la forma $l[x_1 \leftarrow t_1, ..., x_n \leftarrow t_n]\{\alpha_1, ..., \alpha_m\}$, donde l representan la etiqueta de la regla; $x_1, ..., x_n$ representan variables de la regla; $t_1, ..., t_n$ representan los términos concretos a los que se va a instanciar las variables anteriores y $\alpha_1, ..., \alpha_m$ las estrategias con las que se va a llevar a cabo la reescritura en las condiciones de reescritura de la regla condicional. De esta manera, si una regla condicional tiene m condiciones del tipo $p_i \Rightarrow q_i$, cada estrategia α_i controla cada una de estas reescrituras.

La aplicación de reglas con sustitución de términos nos permite trabajar con las reglas definidas con `nonexec`. De esta manera, las reglas que no son admisibles pueden aplicarse en el motor de reescritura de Maude. Por ejemplo, si queremos aplicar la regla *Learn* definida anteriormente, podemos hacerlo a nivel de estrategia instanciando con la cláusula concreta que queremos aprender.

```
Learn[C <- Clause] .
```

Otro tipo de estrategia básica que podemos aplicar son los *tests*. Estas estrategias comprueban si el término sujeto cumple ciertas condiciones, de tal manera que la estrategia mantiene el término actual en caso de cumplirlas, o no devuelve resultado en caso contrario. Los *tests* sirven esencialmente para controlar el flujo de las estrategias, de tal manera que podemos abandonar aquellos caminos que no satisfagan ciertas condiciones. Su sintaxis es la siguiente:

```
match <Pattern> [ s.t. <EqCondition> ]
```

La palabra reservada `match` indica que la comprobación la hacemos con el término entero que estamos considerado. Existen también las palabras reservadas `amatch` y `xmatch` para lograr otros comportamientos, pero en nuestro caso solo nos interesa `match`.

Es importante destacar que el ámbito de las variables que se emplean en un test queda limitado a la declaración del mismo. Es decir, ninguna estrategia posterior puede hacer uso de estos.

Existe también la posibilidad de extender estos *tests*, de manera que se permita reescribir términos dentro del término sujeto. Su sintaxis en Maude es la siguiente:

`matchrew P[`x_1 `, ... ,` x_n `] s.t. C by` x_1 `using` α_1 `, ...,` x_n `using` α_n

donde x_i representa ciertas variables incluidas en el encaje de patrones, y α_i las estrategias que se emplean para reescribir cada una de esas variables. De nuevo, existen sus correspondientes versiones `amatchrew` y `xmatchrew`.

En nuestro caso, utilizaremos esta construcción con frecuencia, pues nos permitirá manejar ecuaciones desde el nivel de estrategia, actualizando la información adicional. Así, por ejemplo, podremos generar la cláusula de aprendizaje a partir de la información del término actual.

A.5.2. Operadores de estrategias

El lenguaje de estrategias permite construir estrategias más complejas a partir de la combinación de una o varias estrategias utilizando distintos operadores. En esta sección, vamos a introducir todos estos operadores. Para ello, asumiremos tres estrategias genéricas α, β, γ:

- one(α) : el operador one indica que solo estamos interesados en un resultado de la estrategia α. De esta manera, al obtener un resultado aplicando α, se detiene la búsqueda de más resultados posibles.

  Esta estrategia tiene sentido aplicarla cuando sabemos que solo existe una solución de la estrategia α sobre el término actual, o cuando solo estamos interesados en obtener una solución cualquiera, sin explorar distintas posibilidades en el árbol.

- not(α) : la estrategia not devuelve el término sujeto si la aplicación de la regla α no ha tenido éxito, o no devuelve resultado en caso contrario.

- α ; β : corresponde a la concatenación de estrategias, de tal manera que primero aplica α y posteriormente se aplica β sobre cada resultado obtenido al aplicar α, obteniendo así como resultado todas las posibilidades al aplicar β.

- $\alpha \mid \beta$: el operador unión evalúa α o β de forma no determinista, devolviendo el resultado de aplicar ambas reglas.

- $\alpha *$: ejecuta la estrategia α cero o más veces consecutivas.

- $\alpha +$: ejecuta la estrategia α una o más veces consecutivas.

- α ! : aplica la estrategia α repetidamente hasta que no devuelve más resultados. Es equivalente a aplicar $\alpha! = \alpha \ * \ ; \ not(\alpha)$.

- α ? $\beta : \gamma$: este operador corresponde al operador condicional, que sigue una construcción parecida a la estructura del if_then_else: si la estrategia α devuelve algún resultado, entonces se comporta como α ; β. En caso contrario, se aplica la estrategia γ con el término sujeto.

- α or-else β : si la estrategia α no devuelve resultado, entonces aplicamos la regla β al término sujeto.

- try(α) : el operador try intenta aplicar la estrategia α, y en caso de que no devolver ningún resultado devuelve el término sujeto inicial.

- test(α) : el operando test nos permite comprobar si se puede aplicar o no la estrategia α, pero devolviendo el término sujeto inicial en caso de tener éxito.

A.5.3. Módulos de estrategias

Como hemos podido observar en secciones anteriores, el lenguaje de estrategias es muy potente de por sí y dispone de un gran número de operadores con los que poder trabajar. Sin embargo, en el momento en el que se empiezan a concatenar varias estrategias distintas, la estrategia resultante es muy larga y pesada para trabajar a nivel de comandos.

Es por ello que los módulos de estrategias resultan muy útiles a la hora de trabajar con ellas. En estos módulos podemos agrupar varias estrategias para formar una nueva, de tal manera que podemos llamarla utilizando la etiqueta con la que se define.

Además, los módulos de estrategias soportan muchas de las estructuras y operaciones que hemos definido previamente, con lo que conseguimos un mayor poder expresivo. Por ejemplo, se pueden incluir parámetros en las estrategias, utilizar encajes de patrones para decidir si una estrategia se aplica o no,

o incluir distintas definiciones de una misma estrategia, de tal manera que se aplica una definición u otra en función de encaje de patrones o condiciones. También se permite la definición de teorías, vistas y parametrización en estos módulos.

En nuestro caso, trabajar con módulos de estrategias nos permitirá llevar información extra que no tiene sentido incluir a nivel de reglas. Cada resolutor SAT almacena información distinta y aplica distintas heurísticas, pero comparten exactamente el mismo sistema de inferencia. Por lo tanto, solo es necesario definir un módulo genérico con esas reglas, e incluir distintos módulos tanto de ecuaciones como de estrategias que consigan el comportamiento deseado.

La declaración del módulo de estrategias es similar a la de los módulos funcionales y de sistema:

```
smod <ModuleName> is <DeclarationsAndStatements> endsm
```

Para declarar una estrategia, primero tenemos que incluir su declaración

`strat slabel :` s_1 `...` s_n `@` s `.`

donde $s_1 ... s_n$ representan los tipos de los parámetros que recibe esa regla, y s especifica el tipo del término sujeto. A esta declaración le sigue una serie de definiciones, que pueden ser o bien condicionales o incondicionales

`sd slabel (` p_1 `, ... ,` p_n `)` $:=$ α `.`
`csd slabel (` p_1 `, ... ,` p_n `)` $:=$ α `if` $C_1 \wedge \ ... \ \wedge C_n$ `.`

En este caso, las condiciones son las mismas que las descritas en un módulo funcional en la sección A.2, no pudiéndose incluir condiciones del tipo $p \Rightarrow q$.

En este apartado, no hemos incluido ejemplos relevantes, pues para entender las estrategias que hemos incluido en este trabajo, necesitamos tener más nociones sobre el problema del SAT y cuál suele ser la estrategia que siguen la mayoría de resolutores. Durante los siguientes capítulos, haremos hincapié en la parte de estrategias, explicando cómo hacer uso de ellas para sacar la máxima potencia disponible.